TOWARDS A SCALABLE DESIGN OF VIDEO CONTENT DISTRIBUTION OVER THE INTERNET

A Thesis
Presented to
The Academic Faculty

by

Mungyung Ryu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2014

TOWARDS A SCALABLE DESIGN OF VIDEO CONTENT DISTRIBUTION OVER THE INTERNET

Approved by:

Umakishore Ramachandran,
Committee Chair
College of Computing
Georgia Institute of Technology

Umakishore Ramachandran, Advisor
College of Computing
Georgia Institute of Technology

Karsten Schwan
College of Computing
Georgia Institute of Technology

Constantine Dovrolis
College of Computing
Georgia Institute of Technology

Moinuddin K. Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Naresh Patel
NetApp
NetApp

Date Approved: 16 May 2014

To all my loving family,

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

We are witnessing a proliferation of video in the Internet; YouTube is the most bandwidth intensive service of today's Internet. It accounts for 20 - 35% of the Internet traffic with 35 hours of videos uploaded every minute and more than 700 billion playbacks in 2010. Netflix, a web service that streams premium contents such as TV series, shows, and movies, consumes 30% of the network bandwidth in North America at peak time. Recently, leveraging the content distribution networks (CDNs), a new paradigm for video streaming on the Internet has emerged, namely, Adaptive HTTP Streaming (AHS). AHS has become the industry standard for video streaming over the Internet adopted by broadcast networks as well as VoD services such as YouTube, Netflix, Hulu, etc.

In the 90's and early 2000's, Internet-based video streaming for high-bitrate video was challenging due to hardware limitations. In that era, to cover the hardware limitations, every software component of a video server needed to be carefully optimized to support the real-time guarantees for jitter-free video delivery. However, most of the software solutions have become less important with the remarkable hardware improvements over the past two decades. There is 100× speedup in CPU speeds; RAM capacity has increased by 1,000×; hard disk drive (HDD) capacity has grown by 10,000×. Today, CPU is no longer a bottleneck for video streaming. On the other hand, storage bandwidth and network bandwidth are still serious bottlenecks for large scale on-demand video streaming.

In this dissertation, we aim at a scalable video content distribution system that addresses both storage bottleneck and network bottleneck. The first part of the dissertation pertains to the storage system on the server side: A multi-tiered storage system that exploits a flash memory solid-state drive (SSD) can meet the bandwidth needs in a much more cost-effective way than a traditional two-tier storage system. We first identify the challenges in architecting such a system given the performance quirks of flash-based SSDs, and the limitations of state-of-the-art multi-tiered storage systems for video streaming. Armed with

the knowledge of these challenges, we show how to construct such a storage system and implement a real web server with multi-tiered storage, evaluate the system with AHS workloads, and demonstrate significant performance gains while reducing the TCO. The second part of the dissertation pertains to the network system on the client side: Integrating peer-to-peer (P2P) technology with the client-server paradigm results in a much more scalable video content distribution system. AHS is a paradigm for *client-driven* video streaming; its philosophy matches well with that of P2P video streaming. An adaptation mechanism is the most important component of AHS that determines overall video streaming quality and user experience. We show a throughput-smoothing-based adaptation mechanism that is designed for a client-server architecture does not work well for a P2P architecture. We provide a buffer-based adaptation mechanism, evaluate our solution with OMNeT++/INET simulator, and demonstrate significant performance gains.

CHAPTER I

INTRODUCTION

Video streaming is a killer application for the Internet. Video traffic is growing fast and becoming a dominant fraction of the Internet traffic; It is expected that 90% of the Internet traffic will be video in 2017 [39]. Such exploding growth of video traffic is attributed to the growing popularity of video services such as YouTube, Netflix, and Hulu. YouTube accounts for 20 - 35% of the Internet traffic with 35 hours of videos uploaded every minute and more than 700 billion playbacks in 2010. Hulu had more than 30 million unique viewers in the U.S in November 2011 [9]. Netflix currently has 20 million subscribers in the U.S., and they consume 30% of North American Internet traffic during the peak time [12].

Adaptive HTTP streaming (AHS) is a new paradigm of video streaming over the Internet currently being used by major content distributors such as YouTube, Netflix, and Hulu. AHS does not depend on specialized video servers. Instead, AHS exploits off-the-shelf web servers. The advantage of this paradigm is that it can easily exploit the widely deployed content distribution network (CDN) infrastructure for a scalable video streaming service, and the firewall and NAT traversal problems can be greatly simplified. On the other hand, its weakness is that there is no QoS mechanism on the server side to guarantee video delivery. It relies on the large resource provisioning (i.e., CPU, RAM, storage capacity, storage and network bandwidth, etc.) that is customary with CDNs. This approach achieves simple system design, scalability, and jitter-free video streaming at the cost of large resource over-provisioning. Therefore, a large scale on-demand video streaming via AHS requires a significant amount of storage bandwidth and network bandwidth for a reliable video delivery.

This dissertation addresses both storage bottleneck and network bottleneck for a large scale video streaming via AHS. Hard disk drives are currently the primary storage devices that store and serve a large video library. A video service provider should deploy increasing number of disk arrays to get more storage bandwidth proportional to the rising number of

viewers and higher video bitrate (i.e., better picture quality). Circa 2014, a single commodity disk's energy consumption is 12 watts. Therefore, a huge number of disk arrays consume a serious amount of energy, and its heat dissipation requires a significant amount of cooling cost. To address such a storage problem, we suggest a multi-tiered storage architecture that exploits NAND Flash Memory Solid-State Drives (SSDs). Circa 2014, a commodity NAND flash memory SSD's energy consumption is 2 watts while its random read latency is 100 times faster than a commodity disk.

Peer-to-peer (P2P) is a promising technology for cost-effective and large scale video streaming on the Internet. Huang et al. and Cha et al. demonstrated that a server's network bandwidth can be saved by around 97% when the P2P technology is incorporated into the client-server video streaming architecture [53, 35]. To overcome the network bottleneck problem, we integrate the P2P streaming technology into AHS. Such an integration raises an important question. Does the throughput-smoothing-based adaptation mechanism that is designed for a client-server architecture work well for the P2P architecture as well? The throughput-smoothing-based adaptation relies on the assumption that client's downlink bandwidth is the only bottleneck and that the servers have sufficient uplink bandwidth. This assumption does not hold in the P2P architecture because clients can be a video source, therefore, clients' uplink bandwidth can become a bottleneck too. We show that the throughput-smoothing-based adaptation does not work well for the P2P architecture using the OMNeT++/INET simulator [14] and suggest a buffer-based adaptation mechanism for a solution.

1.1 Problem Statement

Adaptive HTTP Streaming paradigm relies on the infrastructure of Content Distribution Networks (CDNs), and the video streaming scalability is limited by the amount of storage bandwidth and network bandwidth provided by CDN servers. The exponential increase of video traffic on the Internet and the increasing demand for a higher bitrate video for a better picture quality necessitates proportionally larger storage bandwidth and network bandwidth from CDN servers. This is a serious scalability problem for large scale video

streaming in the future.

1.2 Thesis Statement

Multi-tiered storage with NAND Flash Memory SSDs has the potential to solve the problems of a traditional two-tier storage system. A flash memory SSD can provide a higher storage bandwidth than HDDs for a given cost, consume a fraction of the power of HDDs, and dramatically reduce the cooling needs. However, the capacity per dollar of an SSD is much smaller than that of a HDD. Therefore, it is not cost-effective to replace HDDs entirely with SSDs for the permanent video storage. On the other hand, it is very attractive to architect a multi-tiered storage for video streaming utilizing the flash memory SSD as a caching device between the DRAM and the disks. In addition, a P2P video streaming technology can reduce the workload directed to central servers through cooperative caching among the peer nodes. Thus, peer-assisted adaptive HTTP streaming architecture that integrates the P2P technology is a promising approach for a scalable video content distribution system. With this background, we state our thesis as follows: *"Integrating a multi-tiered storage system and the P2P technology into the CDN architecture would lead to a scalable video content distribution system"*.

1.3 Roadmap

This dissertation is composed of seven chapters. Chapter 2 introduces background knowledge required to understand this dissertation. In Chapters 3 - 5, we first explore storage issues for a large scale video content distribution system in the adaptive HTTP streaming paradigm. Chapter 3 analyzes the workload characteristics of the adaptive HTTP streaming systems and compares performance of hard disk drives and flash memory SSDs for adaptive HTTP streaming. Chapter 4 presents a study on the performance of state-of-the-art multi-tiered storage systems for adaptive HTTP streaming. Through extensive measurements, we analyze the reasons for the poor performance of the systems. Based on the lessons learned from the analysis, we provide design guidelines for a multi-tiered storage system for adaptive HTTP streaming. Chapter 5 proposes our *FlashStream* system that is designed and implemented based on the guidelines and evaluate the system performance and cost-effectiveness

for adaptive HTTP streaming. In Chapter 6, we shift gear to the network issues concerning a large scale video content distribution system. P2P is a promising technology for reducing the network overhead of central CDN servers through cooperative caching among peers. We integrate P2P technology into the adaptive HTTP streaming paradigm and study a novel bitrate adaptation mechanism for the integrated system. Finally, Chapter 7 concludes and presents a few potential avenues of future exploration.

CHAPTER II

BACKGROUND

2.1 Video-on-Demand over the Internet

The target application class for the proposed research is an on-demand video streaming system over the Internet that serves a large number of video requests simultaneously. YouTube [18], Netflix [11], and Hulu [8] are representative services of this application class. YouTube is the most bandwidth intensive service in today's Internet. It accounts for 20 - 35% of the Internet traffic with 35 hours of videos uploaded every minute and more than 700 billion playbacks reported in the year 2010 [47]. Netflix is the largest subscription service for DVD rental and for streaming video on the web. Hulu is a web service that streams premium contents such as news, TV series, and shows. In November 2011, Hulu had more than 30 million unique viewers in the U.S. [9]. Netflix currently has 20 million subscribers in the U.S., and it consumes 30% of North American Internet traffic during the peak time [12].

In this chapter, we will describe the characteristics and system requirements of the video-on-demand (VoD) application.

2.1.1 VoD Storage System

Video data is classified as *continuous media* (CM) data because it consists of a sequence of media *quanta* (e.g., video frames), which is useful only when presented in time. A VoD storage system differs significantly from other types of systems that support only textual data because of two fundamental CM characteristics: (1) Real-time retrieval, and (2) High data transfer rate and large storage capacity. We expand on these characteristics in this subsection.

Because media quanta convey meaning only when presented continuously in time, VoD storage systems must ensure that the playback of each media stream proceeds at its real-time data rate. CM data should be delivered before the data becomes meaningless. Failure to meet this real-time constraint leads to jerkiness or hiccups in video display. To meet the

real-time constraint, the VoD storage systems employ two mechanisms, namely, real-time disk scheduler and admission controller [50]. Traditional systems employ disk scheduling algorithms that aim to minimize seek latency and rotational latency, to achieve high throughput. However, VoD systems use disk scheduling algorithms such as Scan-EDF [85] or Grouped Sweeping Scheme [49] that aim to retrieve data before its deadline. Additionally, VoD systems must employ admission control algorithms to determine whether a new stream can be serviced without affecting streams already being serviced.

A digital video playback demands high bitrate. The size of a video is determined by the bitrate of the video and the length of the video, therefore, a video requires a large amount of capacity from the storage device. In practice, a VoD system must handle playback requests for several streams simultaneously. Even when multiple streams access the same file (such as a popular file), different streams might access different parts of the file at the same time. Therefore, a number of sequential read streams generates random read operations to the storage device. To serve a large number of high bitrate streams simultaneously, the storage device of the VoD system should support a large amount of random read bandwidth. Moreover, the storage device should have a large amount of capacity to store a number of video files that are large in size.

In summary, VoD storage systems should retrieve media data before its deadline, and require both large bandwidth and large capacity for a scalable system.

2.1.2 User behavior on VoD systems

There has been a significant amount of research effort directed towards understanding user behavior on popular VoD services. Dan et al. [41] and Chervenak [36] examined statistics in magazines for video rentals and reports from video store owners. Both studies found that the video popularity distribution is fitted to a Zipf distribution. Acharya et al. [21] analyzed logs of a VoD system deployed in a campus network for education purpose. Authors observed that users preview the initial portion of a video to find out if they are interested. If they like it, they continue watching, otherwise they halt it. In addition, the trace analysis exhibited a high degree of temporal locality, and the video popularity distribution is even

more biased towards popular videos than a Zipf distribution. Yu et al. [96] analyzed logs of a large VoD system deployed by China Telecom covering a total of 1.5 million unique users for a period of seven months in 2004. They derived accurate models of user behavior and video access pattern. First, user accesses follow a clear temporal pattern that the number of users drops gradually during the early morning (12AM - 7AM), while it climbs up to a peak at after work (6PM - 9PM). User arrival rates match a modified form of the Poisson distribution. Second, the average session length is quite short due to users sampling videos by "scanning" through them, and surprisingly, less popular videos actually have longer session length. Third, video popularity matches the Zipf distribution. Fourth, the change of video popularity over time is greatly influenced by external factors such as "recommended videos" and "most popular videos" lists. Cha et al. [35] studied YouTube, the world's largest user generated contents (UGC) VoD service, and Daum UGC, the most popular UGC service in Korea by crawling those sites and collecting meta information about videos. They found a highly skewed video access pattern of users on both services such that 10% of the top popular videos account for nearly 80% of views. This finding implies that caching can be made very efficient because storing only 10% of long-term popular videos, a cache can serve 80% of requests. In addition, authors investigated the potential benefits of a peer-to-peer (P2P) technique in UGC distribution based on real trace. Surprisingly, they found that P2P-assisted distribution can offload server traffic by 98.7%.

Other than the user behavior studies on popular VoD services, researchers recently studied video delivery mechanisms and the quality of service (QoS) of the VoD services [84, 45, 47, 22, 93].

2.1.3 Adaptive HTTP Streaming Paradigm

Adaptive HTTP Streaming (AHS) is a new video streaming paradigm on the Internet. Rather than rely on the dedicated video servers of yesteryears, AHS exploits off-the-shelf web servers for video streaming. Figure 1 illustrates the mechanism of AHS. A *video object* can be in two different forms; a single large file or multiple small *segment* files that have the same play-out duration, typically a few seconds long. Further, there may be multiple

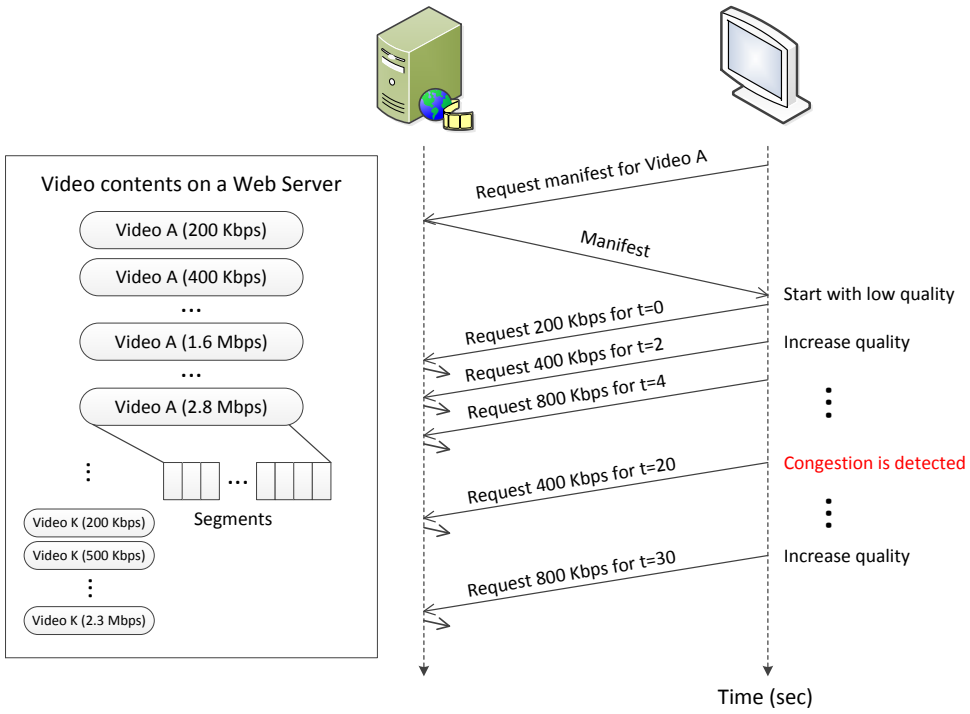


Figure 1: Mechanism of Adaptive HTTP Streaming. A video object is divided into segments, and there are multiple versions of the same video object, supporting different bitrates and different quality levels. Web servers on the Internet that constitute a CDN store these multiple versions of videos. A video player (i.e., a client) can request different segments at different bitrates depending on the state of the underlying network via HTTP.

versions of the same video object, supporting different bitrates and different quality levels. Web servers on the Internet that constitute a CDN store these multiple versions of videos. A player (i.e., a client) can then request different segments at different bitrates depending on the state of the underlying network. If the video object is in the form of a single file, the player requests a segment using the byte ranges protocol of HTTP/1.1 [7]. On the other hand, when the video object is in the form of multiple small segment files, the player requests a segment file download. Notice that it is the player that decides the bitrate to request for any segment. The server treats requests for segments of video files similar to any other normal web request and does not do anything special. This greatly simplifies the server design, and allows the use of existing CDN systems without modification.

In addition to readily exploiting the widely deployed CDN infrastructure for scalable video streaming, AHS can greatly simplify the firewall and NAT traversal problems. On

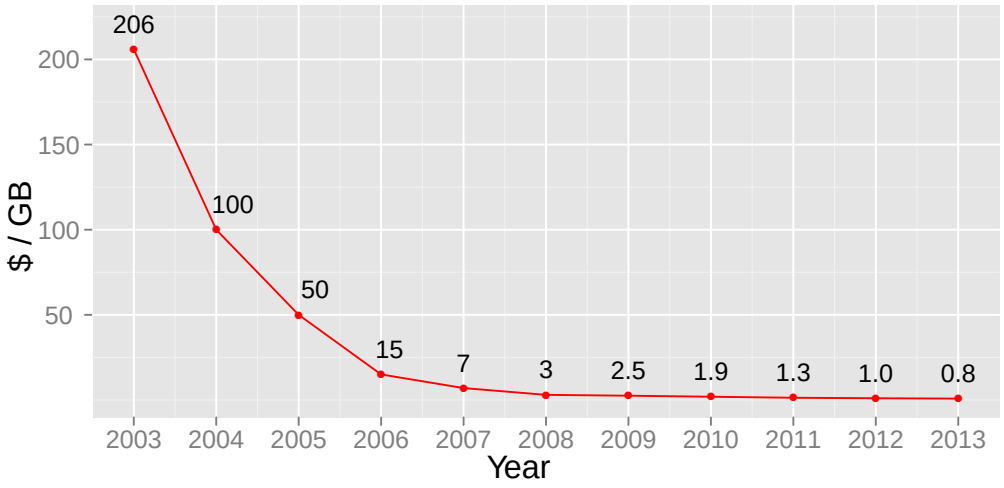


Figure 2: Flash memory $/GB trend

the other hand, its weakness is that there is no QoS mechanism on the server side to guarantee video delivery. It relies on the large resource provisioning that is customary with CDNs. This approach achieves simplicity in system design, scalability, and jitter-free video streaming at the cost of large resource provisioning. Netflix is currently operating their service following the paradigm using various CDNs [22]. Moreover, major software companies recently released their own products adopting the AHS paradigm. Microsoft has *Smooth Streaming*, Apple makes *HTTP Live Streaming* (HLS), Adobe has *HTTP Dynamic Streaming* (HDS). Dynamic Adaptive Streaming over HTTP (DASH) is a standard developed under MPEG, and it became an international standard in November 2011 [10, 90].

2.2 Storage Devices

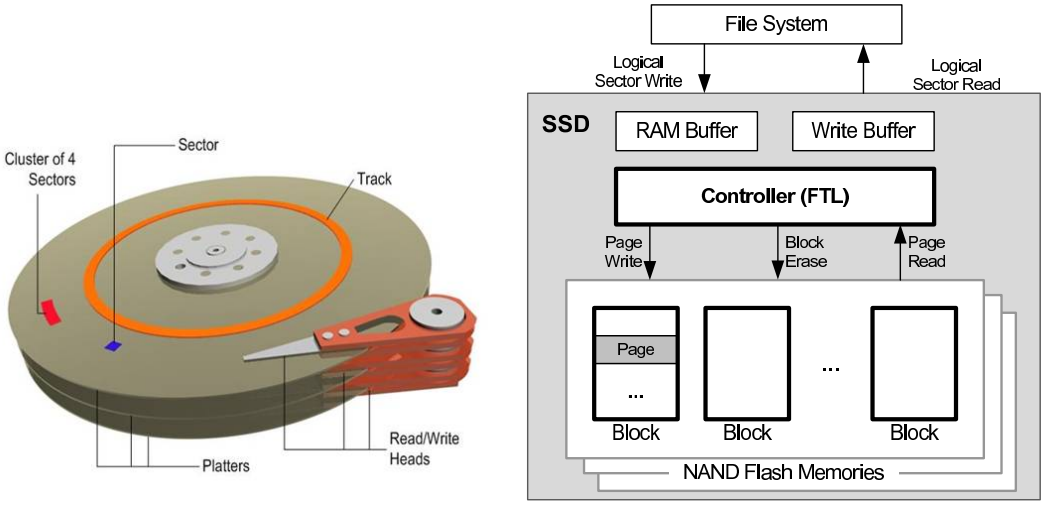
A hard disk drive (HDD) has long been the dominant storage device in a computer system to store and retrieve digital data by virtue of continuously improving cost per unit storage. However, new storage technologies like flash memory and phase change memory are rising competitors threatening HDDs. Solid-State Drive (SSD) is a new storage device that is comprised of semiconductor memory chips (e.g., DRAM, Flash memory, Phase change memory) to store and retrieve data rather than using the traditional spinning platters, a motor, and moving heads found in conventional magnetic disks. The term "solid-state" means there are no moving parts in accessing data on the drive. Among various types of

SSDs, NAND flash memory SSD nowadays is rapidly penetrating into modern computer systems. NAND flash memory densities have been doubling since 1996 consistent with Hwang's flash memory growth model, and it is expected to continue until 2012 [57]. Figure 2 shows that the cost trend of the flash memory conforms to his estimation [69]. Flash memory is originally designed to be used for mobile devices that require low energy consumption, low heat generation, and shock resistance. However, the sharply dropping cost per gigabyte is what has brought NAND flash memory SSDs to the forefront in recent years. Flash-based storage devices are now considered to have tremendous potential as a new storage medium for enterprise servers [51].

In this section, we will explain internal architecture and characteristics of the two popular storage devices (i.e., HDDs and Flash memory SSDs).

2.2.1 Hard Disk Drive

A HDD consists of one or more rapidly rotating discs (i.e., platters) coated with magnetic material, magnetic heads arranged to read and write data on the disk surface, and a moving arm that moves the heads to a target position. Figure 3(a) depicts the internal architecture of a HDD. Since 1956 when HDDs are introduced by IBM, they have been the dominant secondary storage. They have maintained this position through technological advances in capacity, reliability, and speed. Access latency of an HDD consists of seek latency and rotational latency. To access data on the surface of the platters, an HDD should first move a head arm to the location of data. The head arm is a mechanical part and thus it has been challenging to make the seek latency faster. Faster access latency of an HDD is achieved by rotating platters faster, but it adversely increases power consumption, heat generation, and vibration that can jeopardize the reliability of HDDs. Due to such reasons, HDD access latency has *only* improved four times while HDD capacity has grown by 10,000 times over the past two decades [6]. Average access latency of latest HDDs is 3 - 8 ms and energy consumption is 10 - 15 watts. The competitive cost per unit storage of HDDs have made them the dominant storage device, but new competitors like flash memory that have much faster speed, lower energy consumption, and lower heat generation, are threatening the



(a) A HDD consists of spinning platters, a moving arm, and magnetic heads to read and write data

(b) SSD, FTL and NAND flash memory. FTL emulates sector read and write functionalities of a hard disk allowing conventional disk file systems to be implemented on NAND flash memory.

Figure 3: Architecture of a HDD and a Flash Memory SSD

HDD's position.

2.2.2 NAND Flash Memory SSD

Flash memories, including NAND and NOR types, have a common physical restriction, namely, they must be erased before being written [23]. In flash memory, the amount of electric charges in a transistor represents 1 or 0. The charges can be moved both into a transistor by write operation and out by an erase operation. By design, the erase operation, which sets a storage cell to 1, works on a bigger number of storage cells at a time than the write operation. Thus, flash memory can be written or read a single page at a time, but it has to be erased in an *erase block* unit. An erase block consists of a certain number of pages. In NAND flash memory, a page is similar to a HDD sector, and its size is usually 2 to 4 KBytes, while an erase block is typically 128 pages or more. Unless otherwise mentioned, *flash memory* refers to NAND flash memory in this dissertation.

Flash memory also suffers from a limitation on the number of erase operations possible for each erase block. The insulation layer that prevents electric charges from dispersing may be damaged after a certain number of erase operations. For SLC NAND flash memory, the expected number of erasures per block is 100,000 but is only 10,000 for two-bit MLC

NAND flash memory.

An SSD is simply a set of flash memory chips packaged together with additional circuitry and a special piece of software called the Flash Translation Layer (FTL) [58, 60, 80]. The additional circuitry may include a RAM buffer for storing meta-data associated with the internal organization of the SSD and a write buffer for optimizing the write performance of the SSD. The FTL provides an external logical interface to the file system. A sector is the unit of logical access to the flash memory provided by this interface. A page inside the flash memory may contain several such logical sectors. The FTL maps this logical sector to physical locations within individual pages [23]. This interface allows FTL to emulate a HDD so far as the file system is concerned (Figure 3(b)).

To avoid erasing and re-writing an entire block for every page modification, an FTL writes data out-of-place, remapping the logical page to a new physical location and marking the old page invalid. This requires maintaining some amount of free blocks into which new pages can be written. These free blocks are maintained by erasing previously written blocks to allow space occupied by invalid pages to be made available for new writes. This process is called *garbage collection*. FTL tries to run this process in the background as much as possible while the foreground I/O requests are idle, but it is not guaranteed, especially when a new clean block is needed instantly to write a new page. Due to random writes emanating from the upper layers of the operating system, a block may have valid pages and invalid pages. Therefore, when the garbage collector reclaims a block, the valid pages of the block need to be copied to another block. Thus, an external write may generate some additional unrelated writes internal to the device, a phenomenon referred to as *write amplification*. Complicated FTL mapping algorithms that require more resources have been proposed to get better random write performance [80]. However, due to the increased resource usage of these approaches, they are used usually for high-end SSDs. Even though high-end SSDs using fine grained FTL mapping schemes can provide good random write performances, the effect of background garbage collection will be a problem considering the soft real-time requirements of VoD server systems.

The advantages of flash-based SSDs are fast random read (0.05 - 0.1 ms), low power

consumption (1 - 2 watts per drive), and low heat generation due to the absence of the mechanical components. On the other hand, its high cost per gigabyte compared to magnetic disks, poor small random write performance, and limited lifetime are major concerns compared to the disks.

Narayanan et al. [78] have analyzed the efficacy of using Flash-based SSD for enterprise class storage via simulation. In addition to using Flash exclusively as the permanent storage, they also study a tiered model wherein the SSD is in between the RAM and the disks. In their study they use enterprise class SSD ($23/GB). Their conclusion is that SSD is not cost-effective for most of the workloads they studied unless the cost per gigabyte for SSD drops by a factor of 3 - 3000. Their results are predicated on the assumption that SSD is used as a transparent block-level device with no change to the software stack (i.e., application, file system, or storage system layers). The results we report in this dissertation offers an interesting counter-point to their conclusion. In particular, we show that the use of inexpensive commodity SSDs as a buffer cache is a cost-effective alternative for structuring a VoD server as opposed to increasing either the disk bandwidth or RAM capacity to meet a given QoS constraint.

2.3 P2P Video Streaming

In a traditional centralized solution for content distribution, large clusters of servers host the VoD content. These servers require substantial capital investment as well as administration, power, cooling, and maintenance. In response, recent attention has focused on using peer-to-peer (P2P) systems to augment or replace these central VoD servers. P2P techniques seek to improve throughput for bulk downloads by having clients download pieces of the file in parallel from multiple sources (peers) who are simultaneously downloading the same file. The potential advantages of this approach include cost savings, scalability, and ease of deployment.

P2P streaming technology has emerged as a promising technique for cost-effective and large-scale video streaming on the Internet. Prior studies on P2P streaming generally can be classified into two categories: tree-based overlay multicast [37, 30, 62, 33, 94] and a

mesh-based approach [97, 70, 79, 72, 75, 95].

A tree topology is probably the most natural and efficient structure for video streaming, and most of the earlier proposals have adopted the use of tree for overlay multicast. The End System Multicast [37] was one of first such systems, which proposed to move the multicast functionality from the IP network layer to the application layer due to the deployment difficulty of the native IP multicast. The ESM system demonstrated the feasibility of implementing the multicast functionality at end hosts while keeping the core functionalities of the Internet intact. The tree-based systems, however, suffer from node dynamics and thus is subject to frequent reconstruction, which incurs extra cost and renders to sub-optimal structure.

On the other hand, a mesh-based approach has been proven to work well on an unreliable network like the Internet and with high peer churn, and several commercial systems such as PPLive [56, 16] and SopCast [17] have been successfully launched based on the mesh-based approach and have attracted millions of users. Mesh-based P2P video streaming systems work as follows. Peers have partial knowledge as to the membership who are watching a given video. A node exchanges membership information with its neighbors and expands the awareness of other nodes. A video is divided into multiple chunks. A node selects a subset of the neighbors as partners and exchanges video chunk availability information with one another via gossiping. The chunk availability information tells a node which chunks partners have that the node wants to download. The node establishes multiple data streams in parallel with such partners (i.e., those having chunks the node wants) and downloads the chunks. This strategy essentially creates a mesh topology among overlay nodes, which is shown to be robust and very effective in the presence of node dynamics.

CHAPTER III

STORAGE PERFORMANCE FOR ADAPTIVE HTTP STREAMING

For a Video-on-Demand (VoD) system, hard disk drives (HDDs) have long been used to store and serve video data because of their large capacity per dollar and high throughput for accessing video files that are typically large in size. However, this premise does not hold anymore for adaptive HTTP streaming systems. In this chapter, we will explain our reasoning behind such a claim.

3.1 Segment Size in Adaptive HTTP Streaming

As we have explained in Section 2.1.3, a segment (i.e., a part of a video object) is the access granularity in adaptive HTTP streaming. Such a segment size is highly variable due to the following reasons. In AHS systems, every video segment has the same play-out duration. Though each segment has the same play-out time, the size of each segment is different since the video is encoded with a variable bit rate (VBR). In addition, the segment length can be short or long depending on a video publisher's decision. For example, Microsoft's Smooth Streaming uses 2 second segment length by default and Apple's HTTP Live Streaming uses 10 second segment length by default [32]. The segment size is proportional to the length of a segment. Moreover, a single video object is encoded in multiple bitrates in the AHS systems. Since the segment size is proportional to the bitrate, lower-bitrate segments would

Table 1: DASH dataset.

Name	Bitrates (kbps)	Length	Genre
Big Buck Bunny	50,100,150,200,250,300,400,500,600,700,900,1200, 1500,2000,2500,3000,4000,5000,6000,8000	09:46	Animation
Elephants Dream	50,100,150,200,250,300,400,500,600,700,900,1200, 1500,2000,2500,3000,4000,5000,6000,8000	10:54	Animation
Red Bull Playstreets	100,150,200,250,300,400,500,700,900,1200, 1500,2000,2500,3000,4000,5000,6000	97:28	Sport
The Swiss Account	100,150,200,250,300,400,500,700,900,1200, 1500,2000,2500,3000,4000,5000,6000	57:34	Sport
Valkaama	50,100,150,200,250,300,400,500,600,700,900,1100, 1400,1700,2000,2500,3000,4000,5000,6000	93:05	Movie
Of Forest and Men	50,100,150,200,250,300,400,500,600,700,900, 1100,1400,1700,2000,2500,3000,4000,5000	10:53	Movie

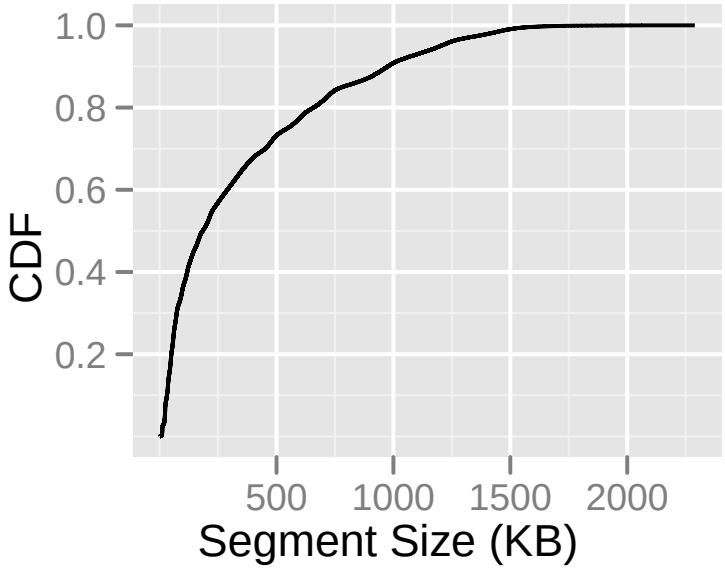


Figure 4: Cumulative Distribution Function (CDF) of segment sizes of DASH dataset for 2 second long segments. 50% of segments are less than 184 KB and the standard deviation is 386 KB.

have a smaller size while higher-bitrate segments would have a larger size. Therefore, the size of segments in the AHS systems may span from a few kilo bytes to a few mega bytes. Dynamic Adaptive Streaming over HTTP (DASH) [10] is an ISO/IEC MPEG standard of the adaptive HTTP streaming paradigm. Table 1 shows DASH dataset that is publicly available for benchmarking.

We have measured the size of segments of DASH dataset in two different segment lengths. Figure 4 shows the cumulative distribution function (CDF) of segment sizes for 2 second long segments. For 2 second long segments, 50% of them have a size less than 184 KB and the standard deviation is 386 KB. Figure 5 shows CDF of segment sizes for 10 second long segments. For 10 second long segments, 50% of them have a size less than 878 KB and the standard deviation is 1865 KB. These graphs tell us that segment sizes could be very small and diverse in the adaptive HTTP streaming. In addition, AHS systems replicate video segments to a number of CDN cache servers for an effective load balancing. A client's request for a segment is directed to a cache server holding the segment via a request routing technique. There are many different request routing techniques such as DNS routing, HTML

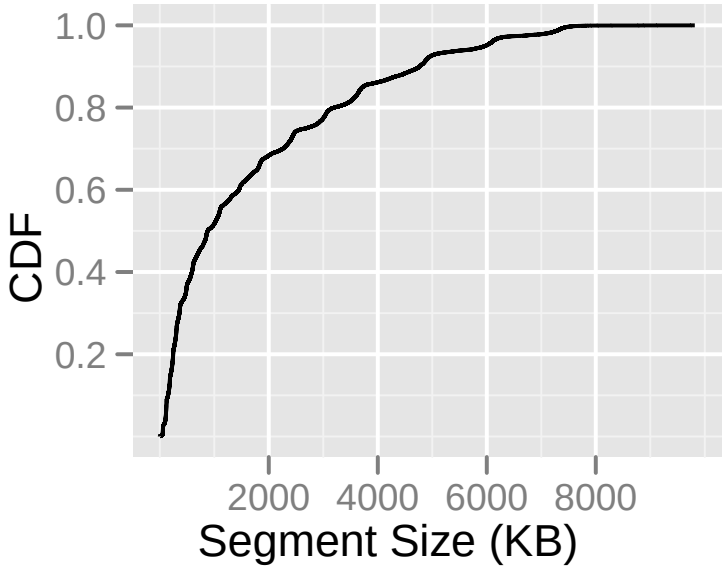


Figure 5: CDF of segment sizes of DASH dataset for 10 second long segments. 50% of segments are less than 878 KB and the standard deviation is 1865 KB.

rewriting [31], and anycasting [81]. For this reason, there is no guarantee that a client who downloaded a segment *i* of a video will download the next segment *i+1* from the same server. The next segment request may be directed to a different cache server that holds a replica. The small and variable segment sizes and the load balancing mechanism of adaptive HTTP streaming systems make a storage system challenging.

3.2 Hard Disk Drive Performance

A hard disk drive (HDD) consists of one or more rapidly rotating discs (i.e., platters) coated with magnetic material, magnetic heads arranged to read and write data on the disk surface, and a moving arm that moves the heads to a target position. Access latency of an HDD consists of seek latency and rotational latency. Seek latency is the time taken to move a head arm to the track of a platter where a target sector belongs to. Rotational latency is the time taken to rotate the platter for the head arm to meet the target sector on the track. Between the two, seek latency is usually longer than rotational latency. In this reason, HDDs are very slow when accessing small data randomly. Figure 6 shows HDD's read throughput in MB/sec against different request sizes. When accessing 4 KB data

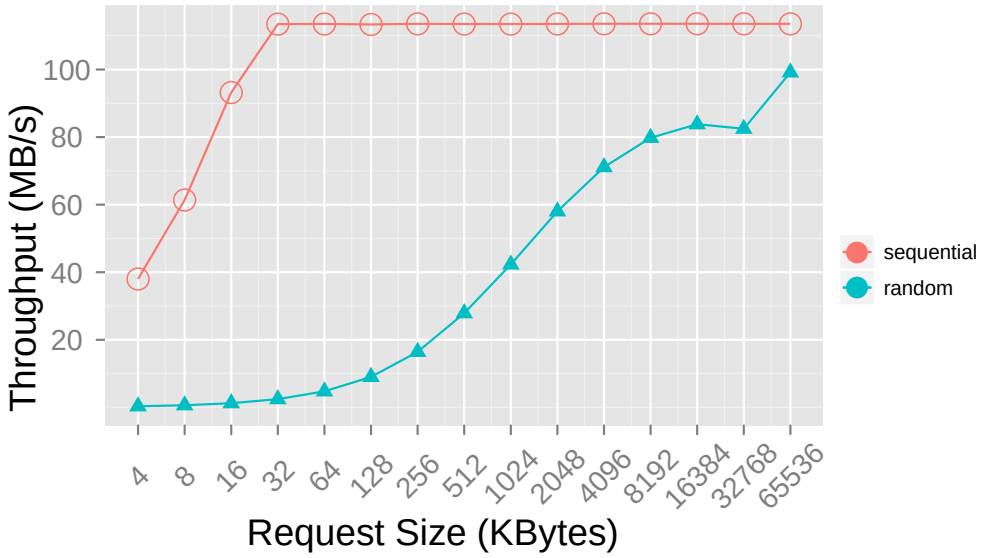


Figure 6: HDD's read throughput against different request sizes. 7200 RPM HDD is used. HDDs show poor performance when reading small data randomly.

randomly, the HDD's throughput is 0.4 MB/sec while the random access of 64 MB data shows 99 MB/sec throughput. In the previous section, we have observed that segment file sizes are very small and diverse in adaptive HTTP streaming. In addition, a CDN cache server receives random segment requests from clients due to the load balancing mechanism of adaptive HTTP streaming systems. In the previous section, we have identified that the median segment size of DASH dataset is 184 KB when the segment length is 2 seconds. With 184 KB request size, the random read throughput of the HDD is very low (approximately 15 MB/sec). Therefore, HDDs are not an ideal storage device for streaming video segments in adaptive HTTP streaming systems.

3.3 Flash Memory SSD Performance

Solid-State Drive (SSD) is a new storage device that is comprised of semiconductor memory chips (e.g., DRAM, Flash memory, Phase change memory) to store and retrieve data rather than using the traditional spinning platters, a motor, and moving heads found in hard disk drives. Among various memory technologies, NAND Flash memory is the technology that is widely used for SSDs nowadays. When we say Flash memory SSDs in this dissertation, it means NAND Flash memory SSDs. One advantage of flash-based SSDs compared to HDDs

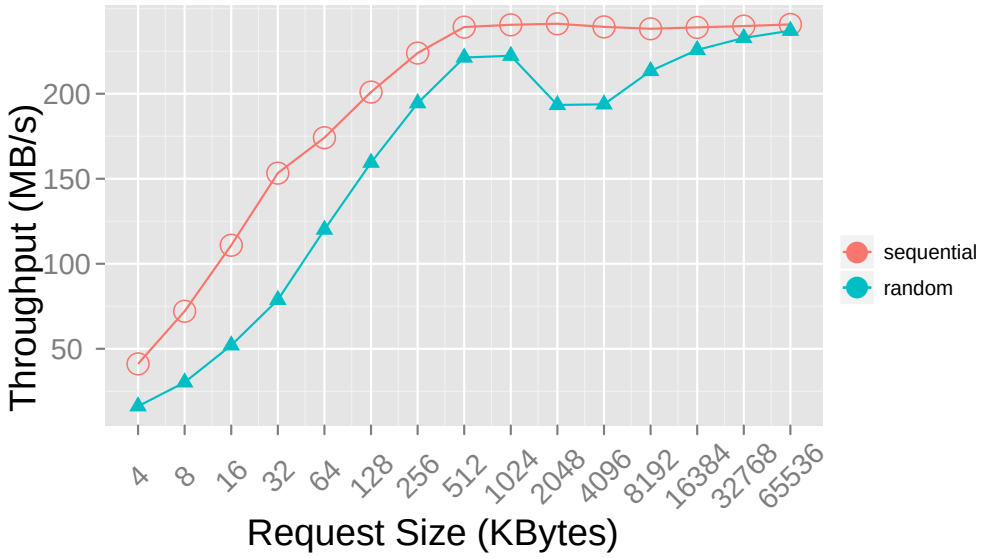


Figure 7: SSD's read throughput against different request sizes. Intel X25-M G1 is used. SSDs show very good performance for random reads.

is its fast random read (0.05 - 0.1 ms). Figure 7 shows SSD's read throughput in MB/sec against different request sizes. The median segment size of DASH dataset is 184 KB when the segment length is 2 seconds (refer to Section 3.1). With 184 KB request size, the SSD can provide around 210 MB/sec throughput for random reads while HDD's random read throughput is approximately 15 MB/sec. Therefore, a flash memory SSD is a promising storage device for adaptive HTTP streaming.

3.4 Summary

In adaptive HTTP streaming systems, a video object is accessed at the granularity of a segment. Such a segment is encoded in multiple bitrates (from low to high bitrate), and the segment length could be short (1 - 2 seconds). Therefore, the size of segments in the AHS systems may span from a few kilo bytes to a few mega bytes. In addition, the segments are accessed randomly in the CDN cache servers. We identified that the median segment size of DASH dataset is 184 KB when the segment length is 2 seconds. With 184 KB request size, the random read throughput of the HDD is very low (approximately 15 MB/sec). On the other hand, flash-based SSDs can provide very large random read throughput (approximately 210 MB/sec) for streaming such small segments. Therefore,

hard disk is not necessarily an ideal storage device for streaming videos using the adaptive HTTP streaming paradigm.

CHAPTER IV

MULTI-TIERED STORAGE SYSTEMS

A large scale VoD system requires a number of HDDs both for capacity (to store the video library) and for bandwidth (to serve the video library). While the cost per gigabyte of HDDs has decreased significantly, the cost per bits-per-second of HDDs has not. Moreover, an array of HDDs consumes a lot of power (approx. 5-15 watts per drive) and generates a large amount of heat; therefore, more power is required for cooling a data center hosting a large array of disks. The amount of user generated content (UGC) and the corresponding number of viewers are increasing explosively on the web. In addition, the AHS approach requires storage bandwidth over-provisioning for reliable video streaming service; therefore, the cost of storage for a large scale service is significant.

Flash memory SSDs open up new opportunities for providing a more cost-effective solution for a VoD storage system. The advantages of flash-based SSDs are fast random read (0.05 - 0.1 ms), low power consumption (1 - 2 watts per drive), and low heat generation due to the absence of the mechanical components. Though flash-based SSDs are attractive as an alternative to HDDs for video storage for all the above reasons, the cost per gigabyte for SSD is still significantly higher than HDDs. Moreover, despite the increasing affordability of SSDs, the ratio of capacity costs of SSD to HDD is expected to remain fairly constant in the future since the bit density of HDDs is still continuously improving. Therefore, a viable architecture is to use the flash-based SSDs as an intermediate level between RAM and HDDs for caching hot contents. Such a storage architecture is called a multi-tiered storage system.

Flashcache [5] and ZFS [19] are state-of-the-art commercial multi-tiered storage systems incorporating flash memory SSDs. Flashcache, developed by facebook, is a write-back persistent block cache designed to accelerate reads and writes from slow storage like HDDs by caching data in faster storage like SSDs. ZFS [19] is a file system that has an intermediate

layer to serve as a read cache between the RAM and the HDDs, called L2ARC. Storage devices that are faster than HDDs are used for L2ARC devices. Though not a requirement, flash memory SSDs can be used as L2ARC devices. We measured the two state-of-the-art flash-based multi-tiered storage systems, and both showed quite disappointing performance for adaptive HTTP video streaming workloads. The poor performance of flashcache and ZFS is very surprising since both systems are designed with flash memory SSDs in mind. In this chapter, we analyze the reasons behind the poor performance of flashcache and ZFS. Based on the lessons learned from the analysis, we provide design guidelines for a multi-tiered storage system for adaptive HTTP streaming. Unless otherwise mentioned, a cache refers to a flash memory SSD rather than RAM in this chapter.

4.1 Measurement

We measured the performance of 3 different storage configurations of an HTTP streaming server: HDDs only, Flashcache, and ZFS. Apache [1] is used for a web server. We implemented a workload generator that emulates a large number of concurrent AHS clients. To measure the storage subsystem performance exactly and to avoid network subsystem effects, the Apache web server and the workload generator run on the *same* machine communicating via a loop-back interface. The machine has Xeon 2.26 GHz Quad core processor with 4 GB RAM, and Linux kernel 2.6.32 is installed on it. We used two 7200 RPM HDDs striped per Linux's software RAID-0 configuration. Intel X25-M G1 and OCZ Core V2 are flash memory SSDs that are used in experiments, but we will show results with only Intel X25-M G1 because results with OCZ Core V2 is similar. Unless otherwise noted, *cache* refers to the flash device used as an intermediate level in the storage hierarchy and not RAM.

4.1.1 Workload

Zipf distribution is generally used in modeling the video access pattern of a video-on-demand (VoD) system, and typically a parameter value between 0.2 and 0.4 is chosen for the distribution [77]. We used 0.271 for the parameter. Dynamic Adaptive Streaming over HTTP (DASH) [10] is an ISO/IEC MPEG standard of the adaptive HTTP streaming paradigm. We use a DASH dataset [65] for our test which is available in the public domain. Among

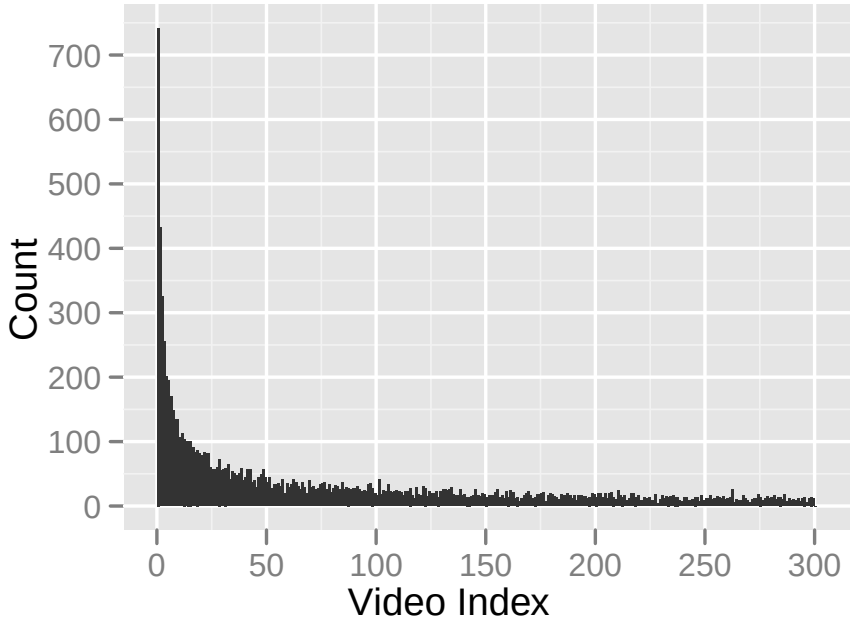


Figure 8: Distribution of access frequency of 300 videos. The zipf parameter is 0.271. Top 60 (20%) popular videos account for 57.6% of total requests.

videos in the dataset, we use *Valkaama* for a test video sequence in this study. The video object is segmented into 10-second long segments, the total length of the video is about 78 minutes, the size is 1.06 GB, and the average bitrate is 2 Mbps. The video object is composed of 466 segment files. Though each segment has the same play-out time, the size of each segment is different since the video is encoded with a variable bit rate (VBR). We copied the video, and make 300 distinct video objects. Figure 8 shows the zipf distribution of the 300 videos.

In every t seconds (which is the inverse of the request rate), the workload generator selects a video object according to the zipf distribution. Next, it chooses a segment of the video object according to the uniform distribution; each segment of the video object has the same probability. The reason for using a uniform distribution is as follows. A large scale HTTP video streaming service like Netflix relies on the CDN infrastructure that widely deploys web cache servers near the edge networks. For an effective load balancing, a video segment (or object) is replicated to a number of web cache servers, and a client's request for the segment is directed to a web server holding the segment via request routing techniques

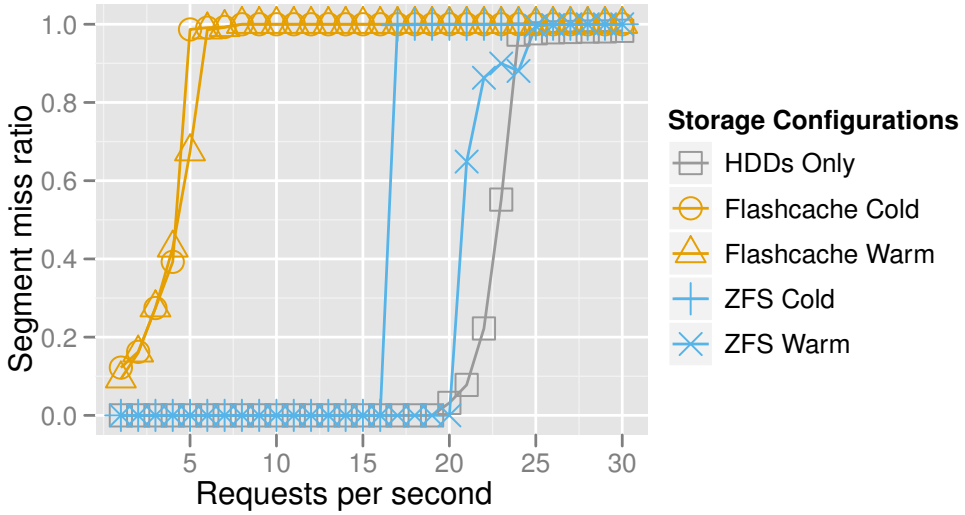


Figure 9: Segment miss ratio as a function of request rate.

such as DNS routing, HTML rewriting [31], or anycasting [81]. For this reason, there is no guarantee that a client who downloaded a segment *i* of a video will download a next segment *i+1* from the same server. The next segment can be served by other web servers that hold a replica. Therefore, it is reasonable to assume a uniform distribution of segment requests to any given web server.

The workload generator sends an HTTP request for the chosen segment to the web server. When the segment is not downloaded to the client within the segment's play-out time (i.e., 10 seconds for our test video), the client counts it as a segment deadline miss. We measured the segment miss ratio against different request rates. *Segment miss ratio* is defined as the ratio of the number of segment deadline misses to the total number of segment requests for a given request rate. Therefore, the *requests per second* is a control parameter, and the segment miss ratio is the measured figure of merit of the storage subsystem.

4.1.2 Measurement Results

As a base configuration for comparison, we use two 7200 RPM HDDs striped per Linux's software RAID-0 configuration. Software RAID is implemented in Linux using the device mapper, which is a Linux storage stack infrastructure. The ext4 file system [4] is installed on these RAID-0 disks. We measure the segment miss ratio with different request rates,

and each measurement is run for an hour. Figure 9 shows for this configuration that the system could serve up to 19 requests per second when the required QoS is 0% segment miss ratio. Since the observed average CPU utilization during the measurements was below 10%, we can conclude that storage is the bottleneck beyond 19 requests per second for this configuration.

Flashcache [5], developed by facebook, is a write-back persistent block cache designed to accelerate reads and writes from slow storage like HDDs by caching data in faster storage like SSDs. Flashcache is a block device level solution, therefore, it is very general and can be utilized at different levels of the software stack (e.g., file systems, applications). We create a single logical block device by having flashcache use the two 7200 RPM HDDs striped per RAID-0 configuration and the Intel X25-M SSD. Out of the total 80 GB available in the SSD, 60 GB are used as flashcache for the experiment. We measure the performance in two different cache states; a cold cache and a warm cache. For the cold cache, we flush the cache at the beginning of each measurement. For the warm cache, we run the workload generator until the cache filling rate falls below 100 KB/sec. We filled the cache up to 60% (i.e., 36GB) by running the workload generator with 1 request per second for 12 hours. Each measurement is run for an hour. Flashcache shows surprisingly poor performance. Figure 9 shows that flashcache could not serve even 1 request per second when the required segment miss ratio is 0%. The flashcache performance is the same whether the cache is cold or warmed up.

ZFS [19] is a file system that has an intermediate layer to serve as a read cache between the RAM and the HDDs, called L2ARC. Storage devices that are faster than HDDs are used for L2ARC devices. Though not a requirement, flash memory SSDs can be used as L2ARC devices. ZFS was originally introduced and implemented in the Solaris operating system, but it has been ported to other operating systems such as FreeBSD and Linux. We use the version of ZFS that is ported to Linux for this measurement. ZFS creates a single storage pool using the two 7200 RPM HDDs in RAID-0 configuration together with the Intel X25-M SSD to serve as the intermediate read cache layer (i.e., L2ARC). Like the flashcache experiment, we use 60 GB of the SSD capacity for the L2ARC cache, and measure

the performance in two different cache states; a cold cache and a warm cache. For the cold cache, we flush the cache at the beginning of each measurement. For the warm cache, we have run the workload generator until the cache is filled in full. Each measurement is run for an hour. Figure 9 shows that ZFS with the cold cache could serve up to 16 requests per second when the required segment miss ratio is 0%. On the other hand, when the cache is warmed up, ZFS could serve up to 20 requests per second. ZFS shows a lot better performance than flashcache, but it is still worse than the base configuration (HDDs-only) when the cache is cold and only slightly better after the cache is warmed up.

4.2 Analysis

The poor performance of flashcache and ZFS is very surprising since both ZFS and flashcache are designed with flash memory SSDs in mind. In this section, we investigate the reasons for this result.

4.2.1 Flashcache

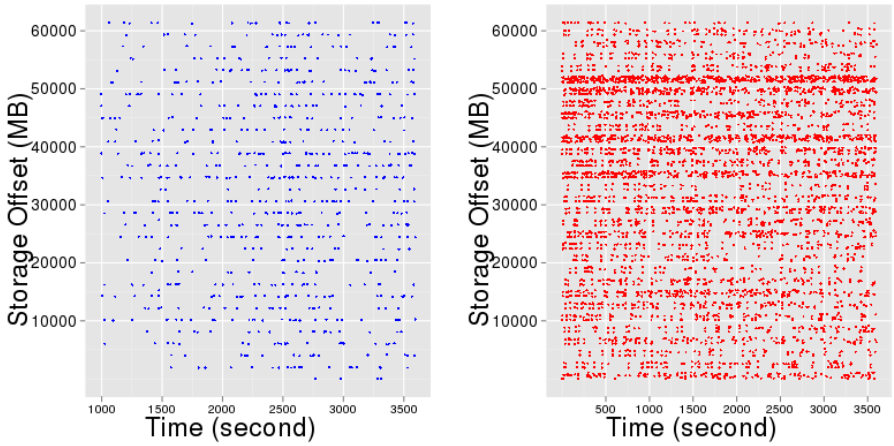
Flashcache organizes the flash memory as a set associative cache. The block size, set associativity, and cache size are configurable parameters, specified at cache creation time. The default block size is 8 sectors (i.e., 4 KB), and the default set associativity is 512 (i.e., a given disk block could be in one of 512 members of a given set) Replacement policy is either FIFO or LRU within a set, and FIFO is the default. We used default values for all the parameters in our flashcache measurements.

In what follows, dbn refers to *disk block number*, the logical device block number in sectors. To compute the target set for a given dbn

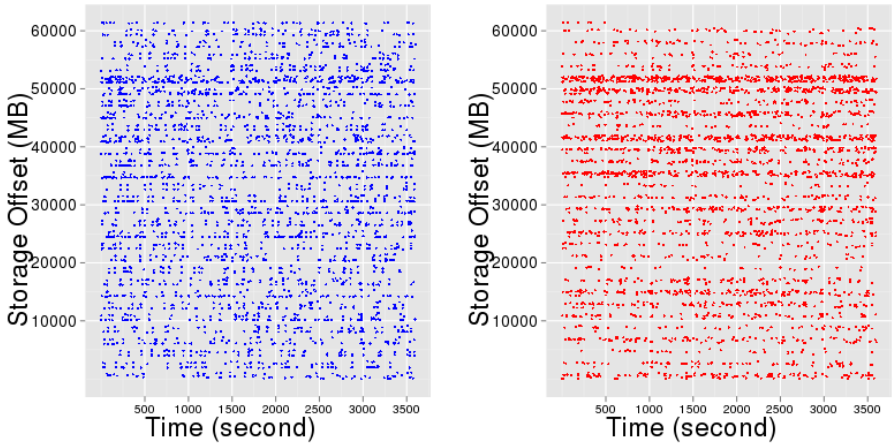
$$target\ set = (\frac{dbn}{block\ size \times associativity})\ mod\ (number\ of\ sets)$$

Once we have the target set, flashcache does a sequential search of the set (linear probing) to find the desired disk block. Note that a sequential range of disk blocks will all map onto a given set. On the other hand, disk blocks that have a dbn difference greater than the associativity will map onto different sets.

When flashcache gets a read request, it calculates the target set from the dbn of the



(a) Cold cache.



(b) Warm cache.

Figure 10: Flashcache's Read (Blue) and Write (Red) access pattern on the cache during 1 hour with 1 request per second. The x-axis is time and the y-axis is the storage offset. Both read and write access patterns are severely random.

request. Then, it probes the cache set to find the requested block. If it finds the block in the cache, it returns the block. Otherwise, flashcache reads the block from the disk, copies the block into the appropriate cache line, and returns the block. A block request that misses in the cache, will first write the block into the cache before returning it to the application. Thus the cache write operation is in the critical path of satisfying an application layer block request that misses in the cache. Therefore, random reads on disks that miss the cache will generate random writes to the cache, and the read could be delayed and miss its deadline if the cache write operation takes a long time to complete.

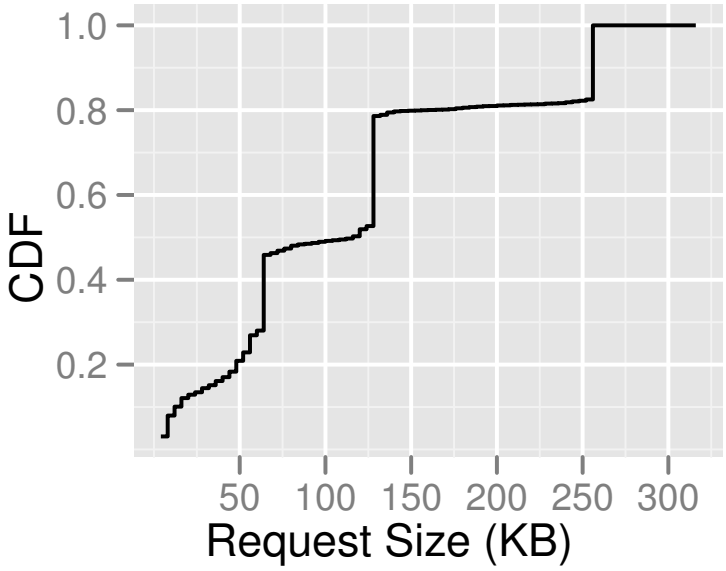


Figure 11: Cumulative Distribution Function (CDF) plotted against the write request sizes sent to flashcache over a period of 1 hour. The median write request size is 116 KB.

Figure 10[1] depicts the read and write access patterns on the SSD for flashcache in two different cache states for an hour period with 1 request per second. This graph is plotted based on the trace data collected using *blktrace* [2], which comes with the Linux 2.6 kernel, to trace the I/O activities at the block device level. To remind the reader, read requests to the flash come from the client requesting video segments that are already cached in the flash; the write requests come from the need to copy blocks from the disk to the flash for requests that miss the flash. The upshot is that both the read and write access patterns are random, and the median write request size is 116 KB (See Figure 11). The request size is defined by the I/O request size sent by the block device driver to the physical block device, which we can determine using *blktrace*. Flash memory shows the best write performance when the request size is a multiple of the erase block size, which is 32 MB for the SSD. A write request size of 116 KB is much smaller than the optimal write request size (i.e., 32 MB). The effect of these small random writes is disastrous on the flash memory performance.

[1]The purpose of this figure is to show that both read and write access patterns for cold and warm cache settings are completely random, as is evident from the distribution of the data points in the scatter plot shown in the figure.

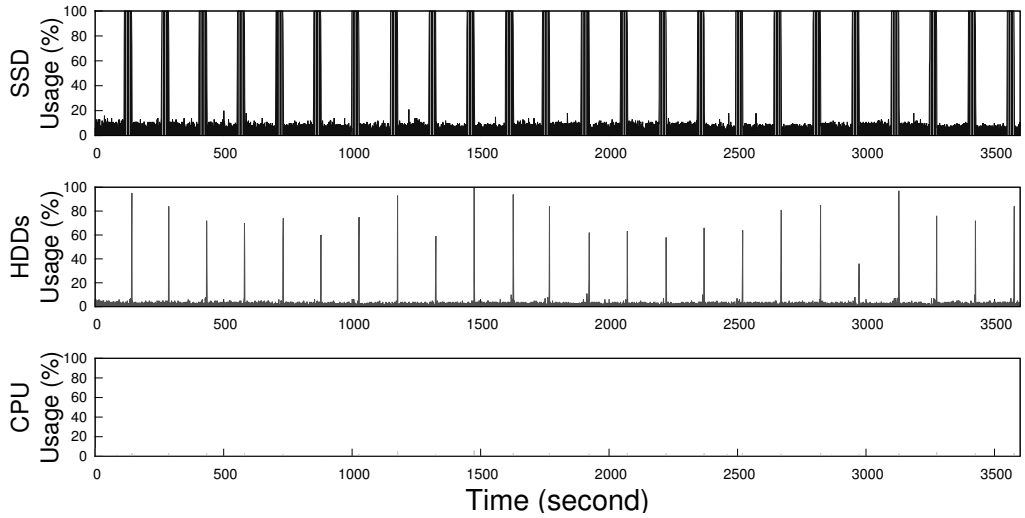
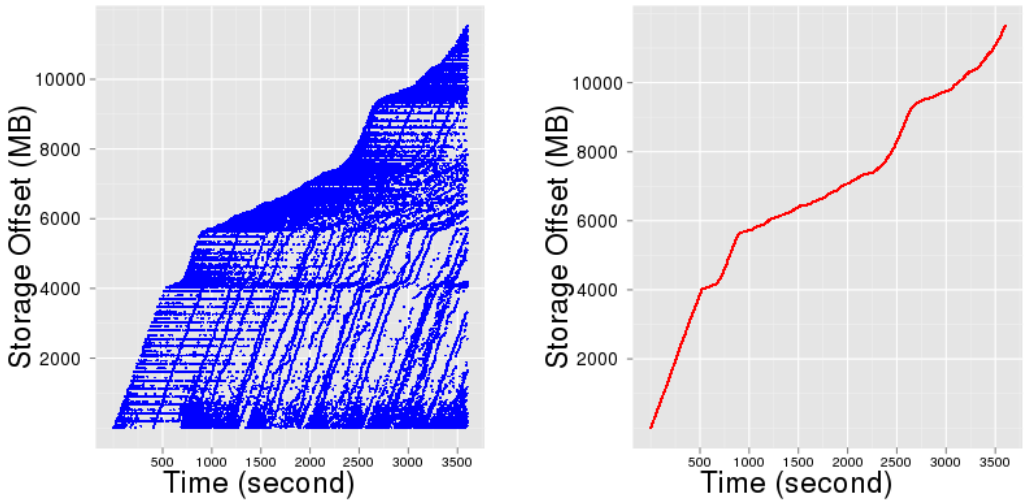


Figure 12: Flashcache's resource utilization during 1 hour with 1 request per second and cold cache. SSD, HDDs, and CPU from top to bottom.

In particular, since flash memory does not support in-place writes, requires erase-before-write, and also requires the erase size be larger than the page size, the small and random write operations will consume fresh pages in a clean block. Ultimately this will lead to the situation where all available clean blocks are used up, thus kick-starting the garbage collection process. Looking at the top chart in Figure 12, which shows the SSD utilization, we can see the utilization peaking up to 100% periodically (around every 140 seconds). It is highly likely that this is due to the garbage collection process. Overall, flashcache generates a very inefficient access pattern for the flash memory SSD.

The read hit ratio of flashcache with a cold cache (Figure 10(a)) is 0.7% while the read hit ratio of flashcache with warm cache (Figure 10(b)) is 25.7%. However, as we have seen in Figure 9, flashcache shows absolutely no difference in performance as measured by the application level segment miss ratio. This can be explained as follows. From the scatter plot for write requests shown in Figure 10(b), we notice that there is a significant volume of write requests to the cache even when the cache is warm. Thus, the application generated reads and system generated writes (a consequence of miss handling) are competing for resources on the SSD device (RAM buffer and CPU). Further, the device processes the requests in order. Thus, if the reads are queued up behind writes, they are likely to miss their deadline even though they hit in the cache since random writes take a long to complete. This is the



(a) Read access pattern.

(b) Write access pattern.

Figure 13: ZFS's Read (Blue) and Write (Red) access pattern on the cache during 1 hour with 16 requests per second and cold cache. The write access pattern is sequential while the read access pattern is random. The median write request size is 128 KB.

most likely reason for the poor application level segment miss ratio observed even with a warm cache in Figure 9 for flashcache.

The reasons for the high segment miss ratio for flashcache while dealing with the AHS workload can be summarized as follows:

1. Upon a cache miss, the block read from the disk has to be first written to the cache before being served to the application. The ensuing small and random write pattern results in long latencies.

2. Second, since flashcache does not give priority to reads over writes to the cache, reads which would be hits in the cache get queued up behind ongoing writes.

Serving the requested segments should be the top priority for a video streaming system. Not respecting this criterion is ultimately the failing of flashcache for this workload.

4.2.2 ZFS

ZFS is smarter than flashcache in handling the flash memory. By analyzing the source code for ZFS L2ARC, we have determined that ZFS basically uses the L2ARC as a FIFO buffer. By definition, the blocks written into the cache are clean (i.e., they are just copies of the disk

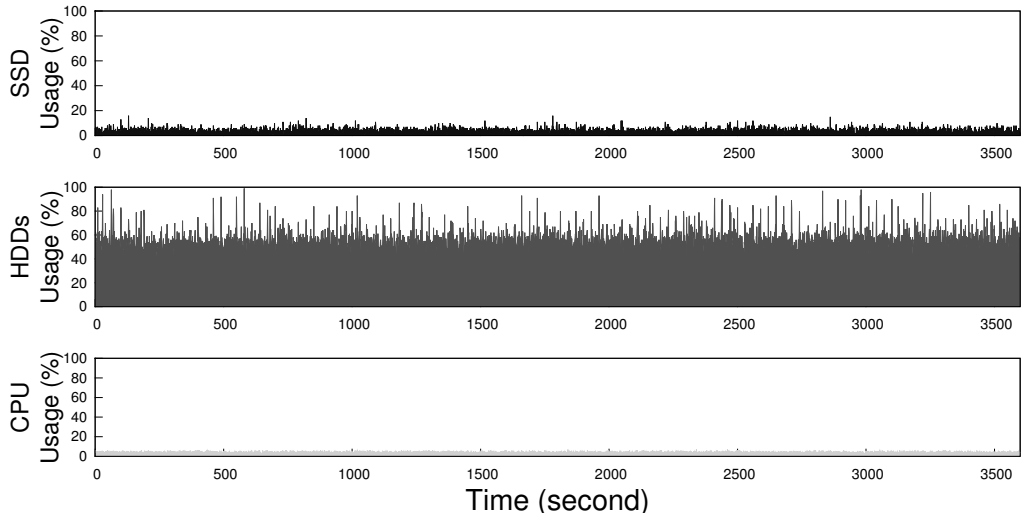


Figure 14: ZFS's resource utilization during 1 hour with 20 requests per second and warm cache. SSD, HDDs, and CPU from top to bottom.

blocks) and therefore they never have to be written back to the disk. ZFS maintains a *write pointer* to the FIFO buffer that L2ARC represents. Once the write pointer reaches the end of the FIFO buffer it simply wraps around to the beginning over-writing the existing blocks in the cache. In other words, ZFS L2ARC converts the random writes to sequential writes. This is evident from Figure 13(b), where it can be seen that ZFS L2ARC sequentially fills up the cache over time.

We have measured the median write size to be 128 KB from the block traces. Interestingly, we have ascertained by executing a micro-benchmark for the SSD that its sequential write throughput peaks at 128 KB. Figure 14 also corroborates our analysis. The top graph in Figure 14 is the activity observed on the SSD. It can be seen that this activity is very low indicating that the SSD performance is not the bottleneck. Thus overall ZFS L2ARC seems to be optimized in handling the flash memory SSDs. That begs the question as to why ZFS performs much worse than the base configuration (HDDs-only) as shown in Figure 9?

The answer is simply due to the fact that L2ARC is being used as a FIFO buffer. In other words, the replacement policy for the L2ARC cache is FIFO. As we noted treating L2ARC as a FIFO buffer is great for write throughput considering the nature of the flash device. However, this leads to a very poor hit ratio. Unfortunately, it is not possible to empirically verify this hypothesis since Linux port of ZFS does not provide statistics such

as L2ARC hit ratio. Therefore, we approximate the hit ratio from the amount of read size requested from the cache and from the disks, which can be obtained by analyzing the block trace. We use following formula to approximate the hit ratio:

$$Hit\ Ratio = \frac{Amount\ of\ read\ on\ SSD}{Amount\ of\ read\ on\ SSD + Amount\ of\ read\ on\ disks}$$

Using this approximation, L2ARC hit ratio is 4.2% and 11.1% for the cold cache and the warm cache, respectively. The low hit ratio is reflected in the low SSD utilization shown in Figure 14.

We have also applied the above formula to flashcache, and the hit ratio is 0.67% and 23.6% for the cold cache and the warm cache respectively. Compare these numbers to the statistics that flashcache provides. They are 0.7% and 25.7% for the cold cache and the warm cache respectively (Refer to Section 4.2.1), therefore, we can say that our approximation for the hit ratio derived from block traces is close to reality.

4.3 Discussion

By studying and analyzing the performance of state-of-the-art multi-tiered storage systems for adaptive HTTP streaming, we have learned a number of lessons. We summarize these lessons in the form of recommendations for constructing a cost-conscious high-performance multi-tiered storage system for AHS.

Recommendation 1: **No small random writes**: Small random writes are extremely inefficient for flash memory SSDs. There are two possible solutions to this problem. First solution is a logging approach. This approach transforms the random writes to sequential writes that the flash memory SSDs can handle very efficiently. However, logging necessitates frequent invocation of the garbage collection anytime it needs to clean obsolete blocks and make room for new data blocks that need to be written. Clearly, this is detrimental to real-time performance such as timely video delivery, since foreground read operations can be stymied and delayed resulting in missing application deadlines. Therefore, logging is not an appropriate solution for the small random write problem in a real-time system like video streaming. A more preferred solution is to write using a much larger granularity. If write operations are requested in multiples of the flash memory's erase block size, and their offset

is aligned with multiples of the erase block size, write amplification will not occur, and flash memory SSDs can handle the writes very efficiently even if the access pattern is completely random.

Recommendation 2: **No flash writes on the critical path**: Writes to the cache while servicing a cache miss should not be in the critical path of serving the requested segment to the application. Hardware caches in a processor are routinely designed in this fashion, wherein the missing data (due to load instruction) is supplied to the processor in parallel with updating the cache. Failing to do this in a video server guarantees that a read request that misses in the cache, will incur the additional penalty of write to the cache when the data is brought from the hard disk. ZFS solves this problem using an evict-ahead policy by which a separate thread copies blocks that are supposed to be evicted soon from RAM to the flash [69, 20].

Recommendation 3: **Higher priority for reads**: Flash reads are more important than flash writes because the former needs to be served before their deadline while the latter is not time critical. Therefore, flash reads should have a higher priority than flash writes when they compete for resources. Both flashcache and ZFS do not consider this point.

4.4 Summary

Due to the cost and size advantage of flash memory compared to DRAM, a multi-tiered storage hierarchy, wherein a flash-based SSD serves as an intermediate level cache appears to be an attractive strategy for constructing a cost-efficient high-performance video streaming server. However, we found through extensive performance studies that two state-of-the-art multi-tiered storage systems exhibited disappointingly poor performance for adaptive HTTP streaming. We performed careful analysis to uncover the sources of poor performance in both these systems. Based on our analysis, we proposed recommendations for constructing a multi-tiered storage system for AHS that avoids the pitfalls in designing such a system.

CHAPTER V

FLASHSTREAM: A MULTI-TIERED STORAGE ARCHITECTURE FOR ADAPTIVE HTTP STREAMING

In the previous chapter, we learned that the current-state-of-the-art in multi-tiered storage systems such as ZFS and flashcache, architected for general-purpose enterprise workloads, do not cater to the unique needs of adaptive HTTP streaming. The lessons learned from Section 4.3 can be summarized as follows:

1. While flash memory SSDs are well-suited to serve the caching needs of videos in an HTTP streaming server, it is of paramount importance to avoid small random writes to the SSDs due to the unique performance characteristic of flash memory.

2. Flash write operations should not be in the critical path of serving missed video segments (brought from the hard disk) to the clients.

3. Flash read operations should have higher priority than flash write operations because the former needs to be served before their deadline to the clients, while the latter is not time critical.

In this chapter, we propose *FlashStream*, a multi-tiered storage architecture incorporating flash memory SSDs that caters to the needs of adaptive HTTP streaming. The unique contributions of our work are the following:

1. To minimize write amplification, writes to SSDs are always requested at the granularity of an optimal block size and aligned on the optimal block size. In addition, we present a novel reverse-engineering technique to find out the optimal block size for any flash-based SSDs. This accomplishes the first design guideline.

2. Utilization-aware SSD admission and ring buffer mechanisms control the write request rate to SSDs and give higher priority to read requests. We adopt ZFS's evict-ahead

policy to avoid SSD write operations from the critical path of serving missed data. This fulfills the second and third design guidelines.

3. Metadata for video segments are embedded in the SSD blocks to quickly restore data in SSD upon power failure.

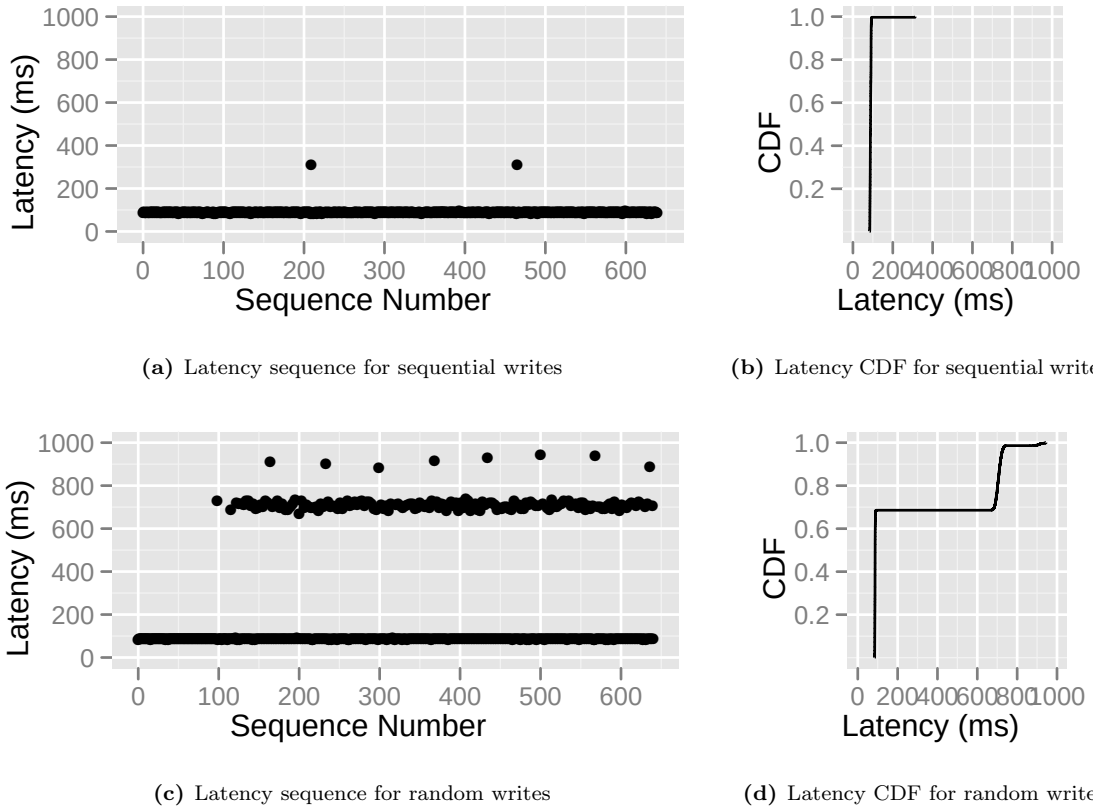
FlashStream is designed and implemented based on the guidelines presented in Section 4.3. We have experimentally evaluated the performance of FlashStream and compared it to both ZFS and flashcache. We show that FlashStream outperforms both these systems for the same hardware configuration.

5.1 Optimal Block Size

As we discussed in Section 2.2.2, the reasons for the poor performance of flash memory for small random writes are two fold: (a) a block has to be erased before a page within it can be written, and (b) an erase block consists of several pages, valid pages in the block being erased must be copied to other clean pages before erasing the block (*write amplification*), and page write and block erasure are inherently slow operations. Due to the write amplification effect, small random writes not only degrade the SSD performance but also reduces the flash memory lifetime.

One way to completely eliminate both "write amplification" and poor performance due to random page writes is to ensure that write requests to the SSD are always in multiples of the block size of the flash memory that is used in the SSD, and that the writes are aligned on block boundaries. This strategy will also have the additional benefit of fast garbage collection (due to the elimination of write amplification) and longer flash memory lifetime. However, there is a catch.

SSD manufacturers put great efforts to fully exploit all available hardware resources and features to improve performance, and they use different designs [23]. In addition, the internal organization of an SSD and their techniques are proprietary and not readily available. How do we choose the right write granularity in the software architecture of the storage system when we do not know the internal architecture of the SSD? To this end, we define the *optimal block size* for write requests to the SSD as the minimum write size that



(a) Latency sequence for sequential writes

(b) Latency CDF for sequential writes

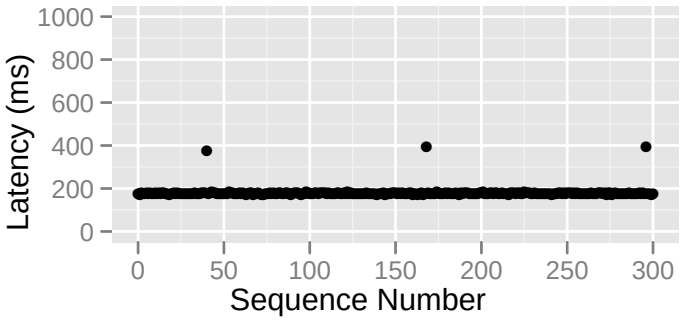
(c) Latency sequence for random writes

(d) Latency CDF for random writes

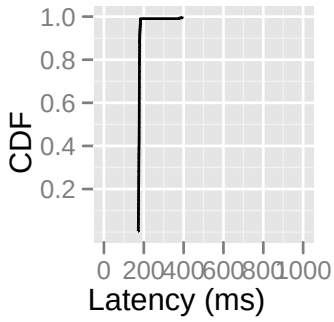
Figure 15: Latency distribution for sequential/random writes with 8 MB request size. OCZ Core V2 is used for the measurement.

makes the random write performance similar to the sequential write performance.

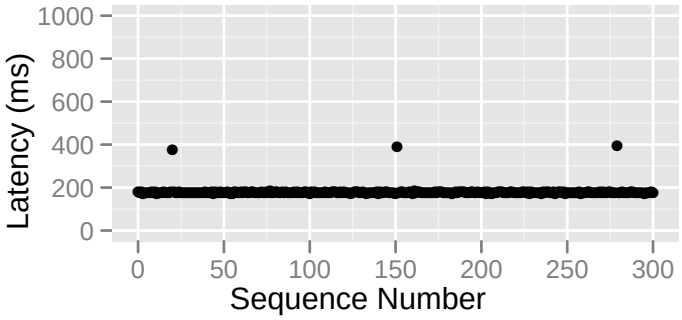
To determine the optimal block size, we do a reverse-engineering experiment on any given flash-based SSD. The idea is to perform random writes with different request sizes until the performance matches that with a sequential workload. To illustrate the technique, we use an OCZ Core V2 flash-based SSD. Figure 15 shows the latency of 640 write operations for sequential and random workloads when the request size is 8 MB on this SSD. For the random writes (see Figure 15(c)), we notice frequent high latencies (i.e., 800-1000 ms) due to the write amplification effect. On the other hand, when the request size is 16 MB (see Figure 16), the latency distributions for the sequential write and the random write workloads are very similar. Figures 15(b) and 15(d) show the CDF graphs for the 8 MB request size; Figures 16(b) and 16(d) show the CDF graphs for the 16 MB request size. While the CDF graphs are useful to visually see the similarity of two distributions, we need a quantitative



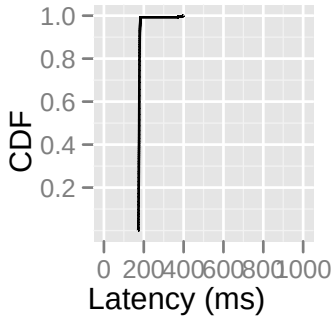
(a) Latency sequence for sequential writes



(b) Latency CDF for sequential writes



(c) Latency sequence for random writes



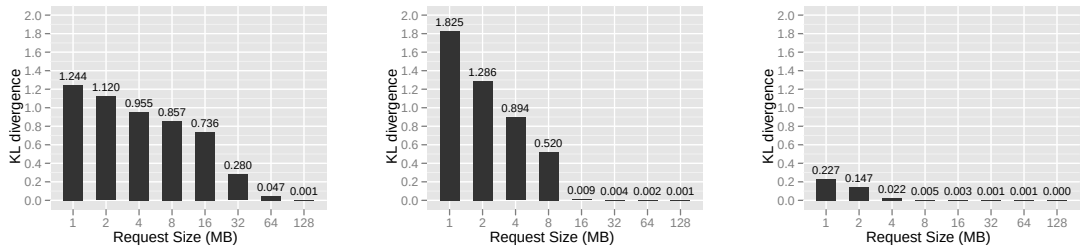
(d) Latency CDF for random writes

Figure 16: Latency distribution for sequential/random writes with 16 MB request size. OCZ Core V2 is used for the measurement.

measure to determine the similarity.

Kullback-Leibler (KL) divergence [64] is a fundamental equation from information theory that quantifies the proximity of two probability distributions: the smaller the KL value the more similar the two distributions. In flash memory SSDs, write operation latency is a random variable and hard to estimate because it is affected by various factors such as mapping status, garbage collection, and wear-leveling. We use the KL divergence value as the metric for comparing the similarity of the latency distributions obtained with the sequential and random write workloads.

Figure 17 shows how the KL divergence value changes with different request sizes for three different SSDs. For all the SSDs, the KL divergence value converges to zero as the request size increases. We let the optimal block size be the request size where the KL divergence value becomes lower than a threshold that is close to zero. The threshold is a



(a) INTEL X25-M G1 **(b)** OCZ Core V2 **(c)** SAMSUNG SLC MCCOE64G5MPP

Figure 17: KL divergence values for different request sizes and SSDs. The optimal block size for INTEL SSD is 64 MB, that for OCZ SSD is 16 MB, and that for SAMSUNG SSD is 4 MB.

configuration parameter. A lower threshold can give a block size that shows more similar latency between sequential writes and random writes. We choose 0.1 for the threshold because Figure 17 shows the KL divergence value quickly converges to zero once it becomes lower than 0.1. Therefore, 64 MB is the optimal block size for INTEL X25-M G1, 16 MB for OCZ Core V2, and 4 MB for SAMSUNG SLC MCCOE64G5MPP. The optimal block size is a function of the internal architecture of the SSD (page size, block size, available internal parallelism for access to the flash memory chips, etc.). However, this experimental methodology allows us to reverse-engineer and pick a write size that is most appropriate for a given SSD. Once the optimal block size has been determined, FlashStream divides the logical address space of flash memory into identical allocation blocks that are of the same size as the optimal block size determined with the experimental methodology.

5.2 System Design

In adaptive HTTP streaming systems, a segment is a basic unit for accessing videos. Accordingly, FlashStream organizes its storage structure at segment granularity. Figure 18 depicts the FlashStream architecture. The first cache layer is a RAM cache. The RAM buffer pool is an array of memory blocks, and each block stores data for a video segment file. Because segments have different sizes, the size of each block in the RAM buffer pool is different. The second cache layer is an SSD cache. The SSD's address space is divided into equal-sized blocks (same as the optimal block size). All write operations to the SSD are requested with the optimal block size aligned on the block boundary. All segment requests

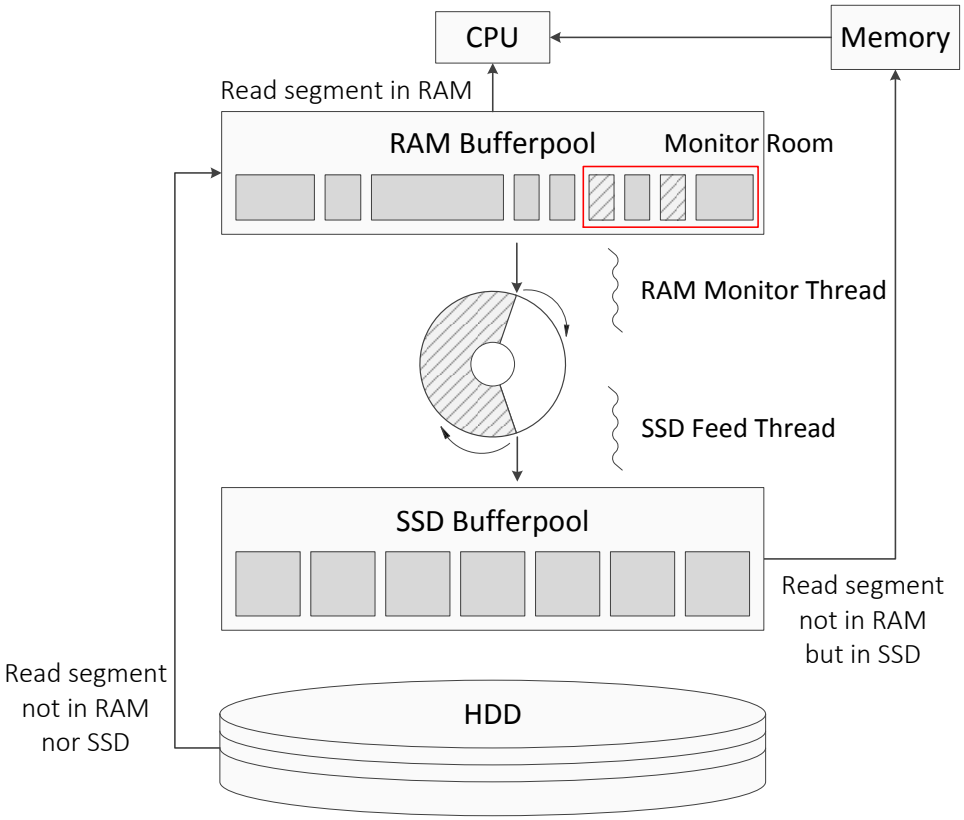


Figure 18: FlashStream Architecture. A finite-sized RAM buffer serves as the first-level cache of the server. SSD is the second-level cache. Upon a miss in the first-level cache, the data is served from the SSD (i.e., it is not copied into the first-level cache). A miss in both the caches results in reading the missing segment from the hard disk and into the first-level RAM buffer cache.

that miss the two caches are served by disks. The segments read from the disks are pushed into the RAM buffer pool, and then they are returned to the CPU. On the other hand, the segments read from the SSD are not inserted into the first-level cache (RAM buffer pool) but they are directly returned to the CPU (i.e., DMA'ed into its memory). In this way, we can maximally utilize the capacity of the first-level RAM cache and the second-level SSD cache because we avoid the same segment from being stored in both levels. One of our design goals is to minimize the I/O requests sent to the disks since the hard disk is the slowest element in the three-level hierarchy of FlashStream, and thus could hurt real-time streaming performance. By storing data either in the DRAM or the SSDs (but not both), FlashStream maximizes the cumulative cache hit ratio in the first two levels and minimizes

the read requests sent to the disks.

5.2.1 RAM Buffer Pool

The RAM buffer pool is the first-level in the three level hierarchy of FlashStream. Segments cached in the buffer are ordered according to the replacement policy used. For example, when a least recently used (LRU) replacement policy is employed, the most recently used segment will be at the head of the buffer pool and the least recently used segment will be at the tail. LRU is the default replacement policy for the RAM buffer pool because it is simple and works well in most cases. However, it should be noted that other replacement policies (such as LRU-Min and GDS [82]) can be easily used for the RAM buffer pool without affecting the design decisions in the other components of FlashStream. Whenever a new segment is inserted into the pool and the pool has insufficient space, then segments at the end of the pool are evicted until the pool has space to host the new segment. A RAM monitor thread periodically monitors the tail part of the RAM buffer pool (Figure 18). This tail portion of the pool is called the *monitor room*. The size of the monitor room is a system configuration parameter, and we use 10% of the total number of segments in the RAM buffer pool for the monitor room size by default. When the thread finds segments in the monitor room that are not in the SSD cache, then those segments are copied to a ring buffer. In Figure 18, such segments are shown as shaded squares in the monitor room. Segments that are in the ring buffer are candidates for being written into the second level cache. The monitor thread simply throws away the segments in the monitor room to make room for new segments coming from the disk even if they have not been copied into the ring buffer by the monitor thread. This is similar to the *evict-ahead policy* used in the ZFS file system. We assume that the video data is immutable and thus read-only (e.g., DB of movies stored at Netflix). When a video object is updated (e.g., new version of a movie is distributed to the CDN servers by the content provider), the cached copies of the segments pertaining to the old version of the video object will simply be removed from the memory hierarchy of FlashStream. This is an out-of-band administrative decision outside the normal operation of FlashStream and does not interfere with the evict-ahead policy for

managing the buffer pool. The advantage of the evict-ahead policy is that it can avoid flash write operations from the critical path for handling cache miss (see Recommendation 2 in Section 4.3). On the other hand, the drawback of the evict-ahead policy is that there is no guarantee that the evicted segments will be written to the second level cache (i.e., SSD). When a segment misses both the RAM and SSD caches, it is read from the disks and placed in the RAM buffer pool. A thread that is inserting a segment into the RAM buffer pool does not have to wait for a victim segment (in the monitor room) to be copied into the ring buffer (for writing eventually to the SSD). The hope is that the victim would have already been copied into the ring buffer by the monitor thread *ahead of time*.

5.2.2 SSD Buffer Pool

Segments that are to be written to the SSD are buffered in the ring buffer between the RAM buffer pool and the SSD buffer pool. While the RAM monitor thread fills up the ring buffer, an SSD feed thread consumes the ring buffer and writes data to the SSD in units of the optimal block size (Section 5.1). In addition, the SSD feed thread employs a simple admission mechanism. For a video streaming system, read operations have higher priority than write operations. Therefore, when the SSD is busy for serving read requests, the SSD feed thread should not make write requests to the SSD. Details of the admission mechanism are presented in Section 5.3.4. If the second-level cache (SSD) is full, the victim block chosen for eviction from the SSD is simply thrown away since the block is already present in the disk, and we are always dealing with read-only data in the storage server.

5.2.3 SSD Block Replacement Policies

A *Least Recently Used (LRU)* replacement policy and a *Least Frequently Used (LFU)* replacement policy are appropriate for evicting blocks from the SSD cache. However, there is an interesting design dilemma. Recall that our unit of management of the second-level SSD cache is the optimal block size of the SSD. Since segments can be of variable sizes, a single block in the SSD cache (of optimal block size) may hold multiple segment sizes. How do we update the LRU list of blocks when a segment that is a small part of a block is accessed? How do we maintain the LFU book-keeping for multiple segments that may be in the same

SSD block? We suggest three different replacement policies:

1. Least Recently Used Block (LRU): In this policy, a segment access is considered the same as an access to the containing SSD block. Therefore, when any segment in a block is accessed, the block is moved to the front of an LRU list. When a block needs to be evicted, the last block in the list is chosen as a victim.

2. Least Frequently Used Block on Average (LFU-Mean): This scheme keeps track of the access count for each segment. When a block needs to be evicted, the average access count of the segments in a block is calculated, and the block with the least average count is chosen as a victim.

3. Least Frequently Used Block based on Median (LFU-Median): Similar to the LFU-Mean policy, this scheme keeps track of the access count for each segment as well. Because the mean is not robust to outliers, this policy uses median of the access counts of the segments in a block.

The performance of these SSD block replacement policies are evaluated in Section 5.4.

5.2.4 Difference from ZFS

FlashStream employs an evict-ahead policy similar to the ZFS file system, however, there are fundamental differences between the two.

1. ZFS uses FIFO replacement policy for the SSD. The motivation is same as FlashStream; FIFO replacement policy generates sequential writes to the SSD and avoids small random writes to the SSD. However, the FIFO replacement policy of ZFS shows a low hit ratio. On the other hand, FlashStream employs different kinds of block replacement policies, and it shows much higher hit ratio than ZFS. The results are presented in Section 5.4.2.

2. ZFS does not differentiate the priority of writes and reads to/from the SSD because it is not designed for a video streaming system. Therefore, reads which would be hits in the SSD cache get queued up behind ongoing writes, and miss the deadline that

reads need to be serviced. On the other hand, FlashStream uses an admission policy for writes to the SSD to give a higher priority for read requests.

3. FlashStream decouples monitoring the RAM buffer and writing evicted segments in the ring buffer to the SSD. These two functions are inter-twined in ZFS, and it could lead to lost opportunity. Recall that the evict-ahead policy used in ZFS will simply throw away pages from the RAM buffer even if they have not been written to the SSD. Thus, if the RAM insertion rate from the disk is faster than the rate at which evicted pages can be written to the SSD, then a larger population of evicted pages will be thrown away. On the other hand, in FlashStream, the monitor thread and the SSD feed thread cooperate through the ring buffer and thus there is an opportunity for more evicted segments to be cached in the SSD (despite the evict-ahead policy) even during periods when the RAM insertion rate is higher than the SSD write throughput. The amount of time taken to fill the SSD cache for both systems is presented in Section 5.4.2.

5.3 Implementation

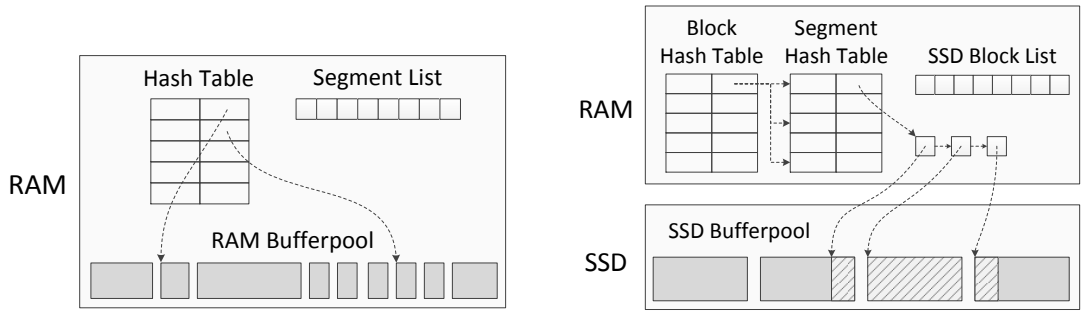
In this section, we describe the details of FlashStream implementation. We have built our FlashStream web server based on *pion-net* [15] open source network library. FlashStream directly manipulates a flash memory SSD as a raw device. In addition, FlashStream bypasses the operating system's page cache and manages the RAM for caching segment files on its own. Each video segment is uniquely identified by a tuple: {video id, segment number, bitrate} (see Table 2). The tuple represents the *id* of a segment. Buffer Manager and SSD Manager are the two most import components, and their details are explained in the following subsections.

5.3.1 RAM Buffer Manager

A buffer manager manipulates the available RAM buffer for caching segments. Figure 19(a) shows data structures used by the buffer manager. It keeps a pool of cached segments in RAM. For fast lookup, a hash table (key, value store) is maintained in RAM. The key is

Table 2: Each video segment is uniquely identified by a tuple: {Video Id, Segment Number, Bitrate}.

Segment Id		
Video Id	Segment Number	Bitrate



(a) RAM Buffer Manager **(b)** SSD Manager

Figure 19: Data structures used by the RAM Buffer Manager and the SSD Manager. Replacement from the RAM buffer is in units of variable-sized video segments.

a segment id, and the value is a tuple: {memory address of the block in RAM, size of the block}. The segment list maintains a list of segment ids for victim selection. The buffer manager uses an LRU replacement policy. A hit in the RAM results in the segment being moved to the head of the segment list. The last element of the segment list is a victim. When a new segment read from the disk is inserted to the RAM buffer pool, a new entry is inserted into the hash table, and the id of the new segment is inserted at the head of the segment list. If needed, a victim is evicted from the buffer pool and the entry for the victim in the hash table is removed.

5.3.2 SSD Manager

An SSD manager manipulates the available SSD blocks for caching segments. Figure 19(b) shows the data structures used by the SSD manager. SSD's address space is divided into equal-size blocks, and the block is same as the optimal block size (see Section 5.1). A block hash table (another key, value store) is maintained to lookup the segments that are stored in a block. The key is an SSD block number, and the value is a list of segment ids that are stored in the same block. A segment hash table is used to lookup the logical block address of the SSD for a segment. When the segment size is larger than the size of an SSD block,

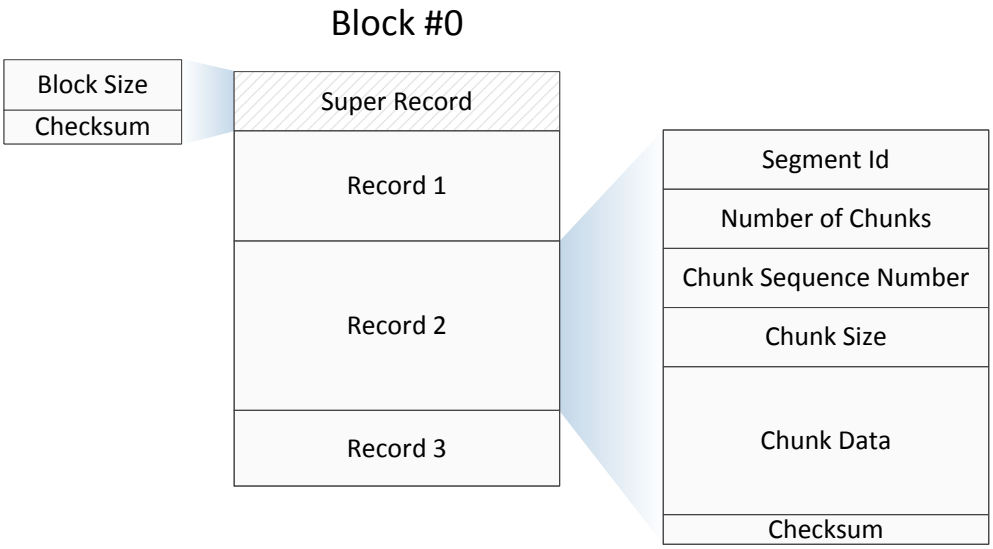


Figure 20: Data structure of an SSD block: An SSD block is composed of records, with each record containing the metadata as well as the data pertaining to the segment id. Block #0 (the first block) of the SSD is special in that it contains a super record in addition to the data records.

then the segment is stored using multiple blocks. In addition, a part of a segment can be written at the end of a block and the rest can be written in the front of another block. We call such partial data of a segment a *chunk* of the segment. Therefore, the value field in the segment hash table is a list of tuples; each tuple contains {SSD block number, offset into the block, size of a segment chunk}. In Figure 19(b), the shaded blocks in the SSD buffer pool represents a single segment stored in multiple SSD blocks. An SSD block list maintains a list of block numbers for victim selection. The block hash table, the segment hash table, and the SSD block list data structures are stored in physical memory (i.e., RAM) for fast access.

5.3.3 SSD Block Data Layout

Flash memory SSDs provide capacity from hundreds of gigabytes to a few terabytes in a single drive these days. After a system crash or administrative downtime, warming up such a large capacity SSD takes a long time. Since flash memory is non-volatile, a warmed up SSD can still serve as a second-level cache despite power failures as long as we have metadata for

the segments stored in the SSD as well. To recover the stored data, FlashStream embeds metadata in the SSD blocks. Figure 20 shows the data layout in the SSD blocks. The first SSD block (block number 0) has a special record, namely, *super record*. The super record includes information about the block size and its checksum. Only the first SSD block has the super record, and the other SSD blocks do not. Following the super record, a series of records are stored. Each record represents a chunk of a segment. The record consists of *segment id*, *number of chunks*, *chunk sequence number*, *chunk size*, *chunk data*, and *checksum*. From the segment id, the system can figure out the segment that the chunk belongs to. The number of chunks tells how many chunks the segment is divided into, and the chunk sequence number tells the order of the chunk in its segment. Checksum is used to verify the integrity of each record. After system crash, by scanning sequentially all SSD blocks, FlashStream can reconstruct the in-memory data structures of the SSD manager (i.e., block hash table and segment hash table).

5.3.4 Utilization-Aware SSD Admission

The SSD feed thread employs an admission mechanism to avoid writes to the SSD when the SSD is busy for serving reads. To measure how busy the SSD device is, we use */proc/diskstats* wherein the Linux kernel reports information for each of the storage devices. */proc/diskstats* provides the amount of time (in milliseconds) spent doing I/O for each storage device. We read the number at the beginning of a certain epoch and again at the end of the epoch. Because this is an accumulated number, the difference between the two readings gives the time spent in I/O for that device for the current epoch. The ratio of the time spent in I/O to the periodicity of the epoch serves as a measure of the storage device utilization for that epoch. The SSD feed thread stops writing data to the SSD when the SSD utilization exceeds a threshold λ, and the default value for the threshold is 60%.

5.4 Evaluation

In this section, we evaluate the performance of the FlashStream system. We would like to mimic the request pattern that is fielded and serviced by an HTTP server (i.e., a CDN server) that is serving video segments to distributed clients under the adaptive HTTP

Table 3: Five system configurations used for our evaluation.

Configuration	Web Server	File System
FlashStream(LRU)	FlashStream	EXT4
FlashStream(LFU-Mean)	FlashStream	EXT4
FlashStream(LFU-Median)	FlashStream	EXT4
ZFS	Apache	ZFS
Flashcache	Apache	EXT4

Table 4: Flash Memory SSDs that are used for our experiments. SSD A shows significantly better random write performance than SSD B. Both SSDs have similar performance for small random reads. On the other hand, the different SSDs have significantly different small random write performance. We intentionally use a very low-end SSD (SSD B) to show how well FlashStream works with low-end SSDs.

	SSD A	SSD B
Model	INTEL X25-M G1	OCZ Core V2
Capacity	80 GB	120 GB
4KB Random Read Throughput	15.5 MB/s	14.8 MB/s
4KB Random Write Throughput	3.25 MB/s	0.02 MB/s

streaming mechanism. To this end, the workload offered to the FlashStream server is a large number of independent requests from clients. To measure the storage subsystem performance exactly and to avoid network subsystem effects, the workload generator program and the FlashStream server run on the same machine communicating via a loop-back interface. The machine has Xeon 2.26 GHz Quad core processor with 8 GB DRAM, and Linux kernel 2.6.32 is installed on it. We use a 7200 RPM HDD, and the capacity of the HDD is 550 GB. We have evaluated five system configurations. In addition to our FlashStream system with three SSD cache replacement policies, we have measured the performance of two state-of-the-art multi-tiered storage systems (that use flash memory caching), namely, ZFS [19] and flashcache [5]. Table 3 shows the web server and the file system that are used for each system configuration. The version of Apache web server is 2.2.14, and the version of ZFS is zfs-fuse 0.6.0-1. The flash memory SSDs we used for our experiments are listed in Table 4. Note that we intentionally use a very low-end SSD (SSD B) to show how well FlashStream works with low-end SSDs. In our evaluation, FlashStream and ZFS have worked slightly better with SSD B while flashcache has worked better with SSD A. Therefore, we present only results of FlashStream and ZFS used with SSD B and results of

Table 5: Source DASH dataset that is used to generate our 2000 videos.

Name	Bitrates (kbps)	Length	Genre
Big Buck Bunny	200, 400, 700, 1500, 2000	09:46	Animation
Elephants Dream	200, 400, 700, 1500, 2000	10:54	Animation
Red Bull Playstreets	200, 400, 700, 1500, 2000	97:28	Sport
The Swiss Account	200, 400, 700, 1500, 2000	57:34	Sport
Valkaama	200, 400, 700, 1400, 2000	93:05	Movie
Of Forest and Men	200, 400, 700, 1400, 2000	10:53	Movie

flashcache used with SSD A. We use only 75 GB out of the total capacity for both SSDs for fair comparison.

5.4.1 Workload

Zipf distribution is generally used in modeling the video access pattern of a video-on-demand (VoD) system, and typically a parameter value between 0.2 and 0.4 is chosen for the distribution [77]. We use 0.2 for the parameter in our study. Dynamic Adaptive Streaming over HTTP (DASH) [10] is an ISO/IEC MPEG standard of the adaptive HTTP streaming paradigm. We use a DASH dataset [65] for our test which is available in the public domain. As shown in Table 5, the dataset comprises six distinct video objects with five different bitrates for each video. However, for our experimentation we needed to create a larger dataset comprising of 2000 distinct videos. In other words, we needed to create a namespace of 2000 distinct videos from the set of 6 videos. We have generated the 2000 videos by uniformly using the six videos and five bitrates. The total size of our dataset is 545 GB. Figure 21 shows the zipf distribution of the 2000 videos. Since we use five different bitrates and videos are encoded with variable bitrate (VBR), the segments of our generated dataset have very different sizes.

In every t seconds (which is the inverse of the request rate), the workload generator selects a video object according to the zipf distribution. Next, it chooses a segment of the video object according to a uniform distribution; each segment of the video object has the same probability. The reason behind the uniform distribution is as follows. A large scale adaptive HTTP video streaming service like Netflix relies on the CDN infrastructure that widely deploys web cache servers near the edge networks. For an effective load balancing,

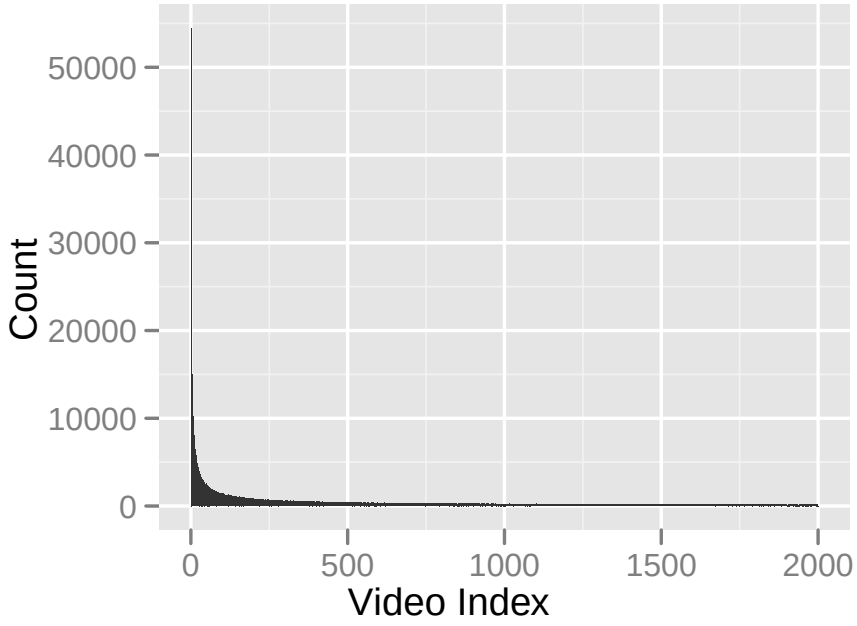


Figure 21: Distribution of access frequency of 2000 videos. The zipf parameter is 0.2. Top 400 (20%) popular videos account for 65.9% of total requests.

a video segment (or object) is replicated to a number of web cache servers, and a client's request for the segment is directed to a web server holding the segment via a request routing technique. There are many different request routing techniques such as DNS routing, HTML rewriting [31], and anycasting [81]. For this reason, there is no guarantee that a client who downloaded a segment i of a video will download the next segment $i+1$ from the same server. The next segment request may be directed to a different web server that holds a replica. Therefore, it is reasonable to assume that the probability that a web server gets a request for a segment of a video object is totally random.

The workload generator sends an HTTP request for the chosen segment to the web server. When the segment request is not satisfied (i.e., it is not downloaded to the client) within the segment's play-out time (10 seconds for our test video), the client counts it as a segment deadline miss. We choose the 10 second segment size because it is the default size used by Apple HTTP Live Streaming [32]. We measure the segment miss ratio against different request rates. *Segment miss ratio* is defined as the ratio of the number of segment deadline misses to the total number of segment requests for a given request rate. Therefore,

the *requests per second* is a control parameter, and the segment miss ratio is the measured figure of merit of the storage subsystem.

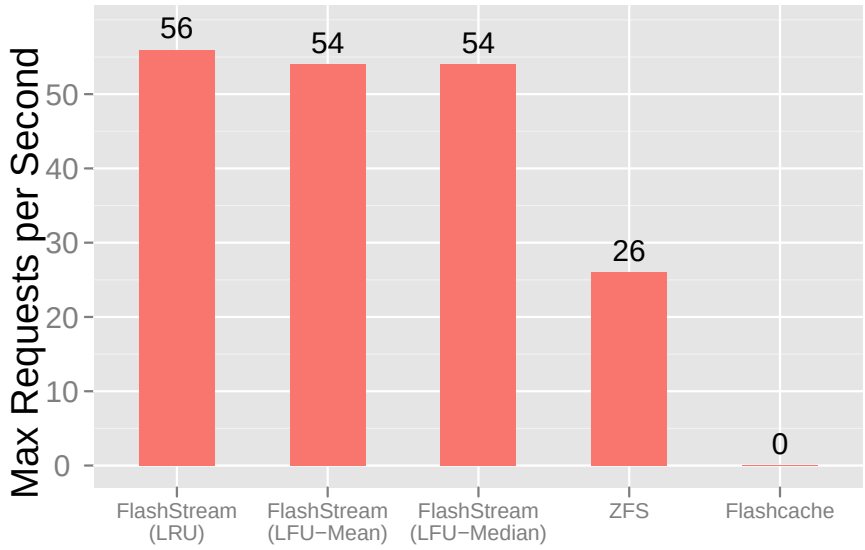
5.4.2 Performance Comparison

In this section, we measure the maximum request rate that the five systems can support when the cache is warmed up. This is the best performance each of the systems can provide. We have fully filled up the SSD cache by running the workload generator for a sufficient amount of time before each measurement. FlashStream (regardless of replacement policy) takes 94 minutes to fully fill up the 75 GB SSD cache while ZFS takes 1200 minutes (20 hours) and flashcache takes 1320 minutes (22 hours) for the same amount of SSD cache. We assume the QoS requirement for video streaming is 0% segment miss ratio. By increasing the request rate gradually, we measure the segment miss ratio for each request rate until the segment miss ratio gets higher than 0%. In this way, we can get the maximum request rate for 0% segment miss ratio. We run each measurement with a different request rate for an hour.

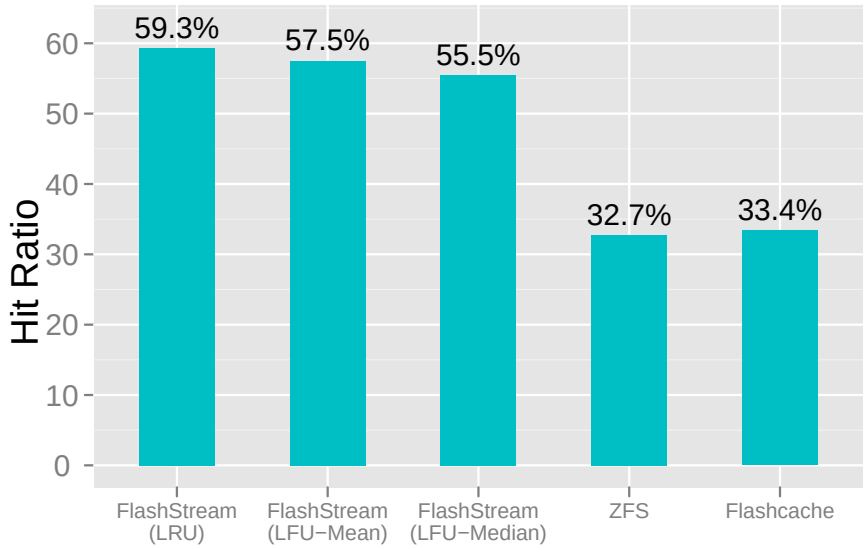
Figure 22(a) shows that FlashStream performs two times better than ZFS. Different SSD cache replacement policies of FlashStream perform similarly while LRU is slightly better than the other two. ZFS is able to support a maximum of 26 requests per second, and flashcache cannot support even 1 request per second.

Next, we measure the SSD cache hit ratio of the five systems with a warmed up cache when the request rate is the maximum request rate that the systems can support. Each measurement is run for an hour. Figure 22(b) shows that FlashStream with LRU SSD cache replacement policy has the best hit ratio. The other two policies (i.e., LFU-Mean and LFU-Median) show slightly lower hit ratio than LRU.

ZFS's lower maximum request rate compared to FlashStream correlates directly with ZFS's lower SSD cache hit ratio. Page reads that miss the SSD cache of ZFS go to disks, makes the disks overloaded, and the reads from the disks miss their deadline. This result tells us, for the SSD cache, block-level replacement policies (that FlashStream uses) are able to achieve a higher hit ratio than FIFO replacement policy (that ZFS uses), and results in



(a) Maximum Request Rate



(b) SSD Cache Hit Ratio

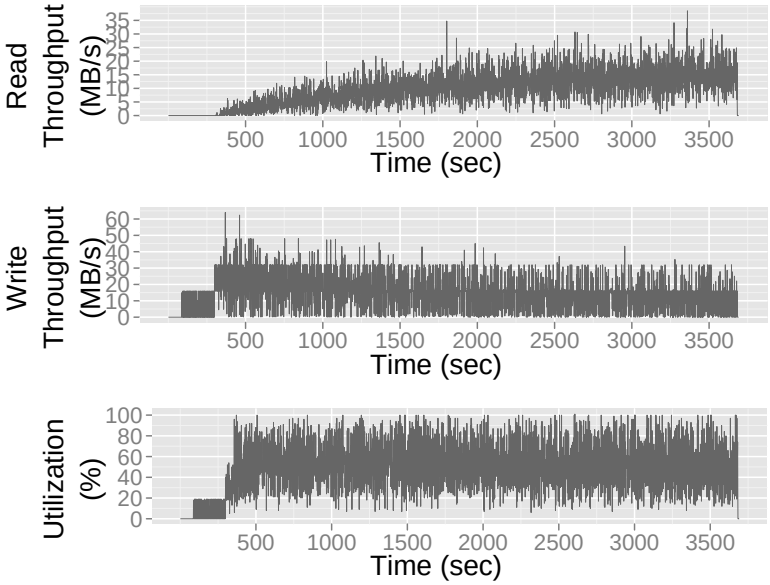
Figure 22: Maximum request rate and SSD cache hit ratio of the 5 different system configurations with a warmed up cache for 0% segment miss ratio. FlashStream performs 2 times better than ZFS.

a higher maximum request rate. The reason for the poor performance of flashcache is due to the fact that SSD write operations are in the critical path of serving read operations that miss the SSD cache. Thus though the observed hit ratio for flashcache is 33.4%, it does not translate to performance (as measured by the maximum request rate) due to the fact that SSD cache reads could be backed up behind long latency SSD cache write operations. FlashStream and ZFS systems address this problem by the evict-ahead mechanism (see Section 5.2.1). For a more detailed analysis of the poor performance of ZFS and flashcache, please refer to Section 4.2.

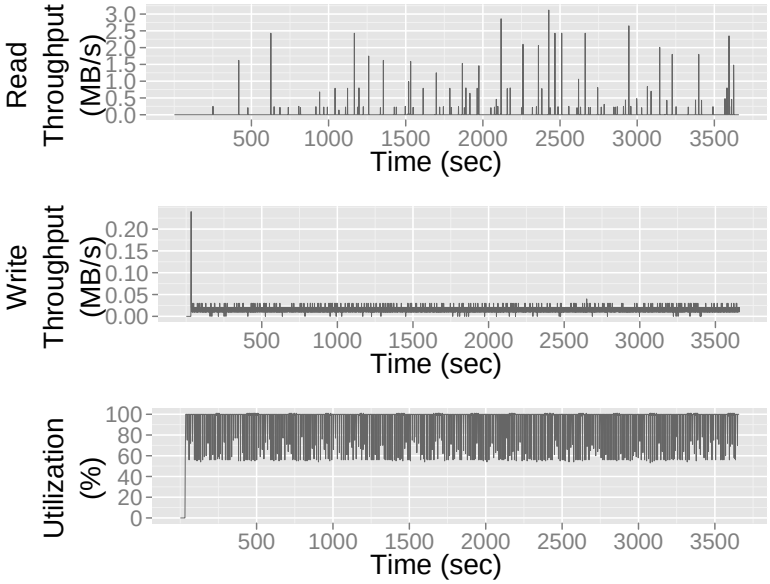
5.4.3 Effect of Block Size

In this section, we measure how the choice of the SSD block size affects the FlashStream performance. We run FlashStream with a cold cache for an hour using SSD B. Even though the SSD cache is empty, for this experiment, we intentionally make FlashStream return random free blocks (not in a sequential order) when the SSD feed thread needs to write a block of data to the SSD. It is because we want to see quickly the effect of the SSD writes with random offsets without fully filling up the SSD cache. This is for a quick measurement and it does not harm the correctness of the results. As we have shown in Figure 17, the optimal block size of SSD B is 16MB. Figure 23(a) shows read throughput, write throughput, and utilization (top to bottom) of the SSD for an hour. From the write throughput graph (middle), we notice that the system gradually fills segments into the SSD cache. The read throughput graph (top) shows that the read throughput increases as segment requests hit the SSD cache. The utilization graph (bottom) tells us that the SSD device is not overloaded during this period.

On the other hand, Figure 23(b) presents the same SSD's read throughput, write throughput, and utilization when the block size is 4KB. SSD B is a low-end SSD that shows very low performance with small random writes (see Table 4). Due to the write amplification triggered by the small block writes with random offsets, the SSD is overloaded during the whole period (100% utilization), and hence provides very low read and write throughput.



(a) 16MB Block Size



(b) 4KB Block Size

Figure 23: Effect of different block sizes on Read throughput, Write throughput, and Utilization of the SSD (second-level cache of FlashStream). When the block size matches the optimal block size (Figure 23(a)), the SSD is not overloaded (i.e., there is no write amplification), and the system fills segments gradually, and serves the segments that hit the SSD. When the block size is 4KB (Figure 23(b)), the SSD is overloaded (due to write amplification), and shows very low read and write throughput.

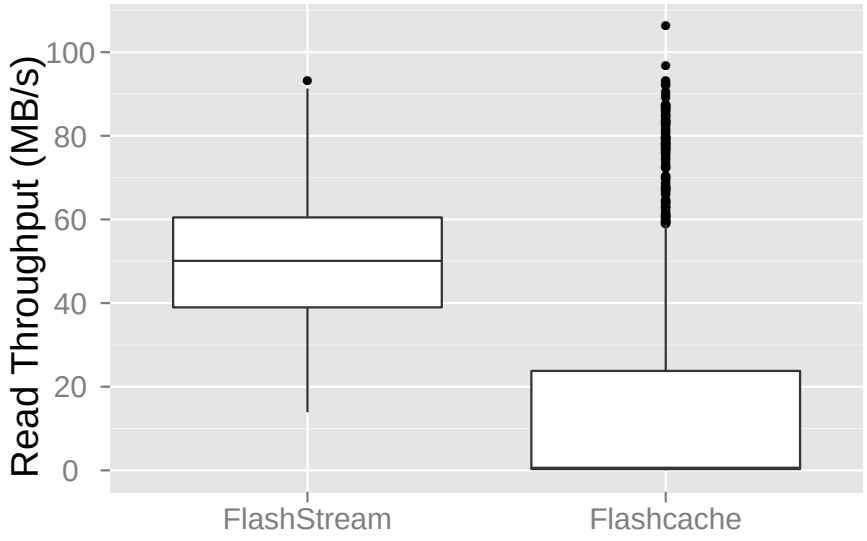


Figure 24: Read throughput variance. FlashStream has a median 50.09 MB/s and a standard deviation 13.68 MB/s. Flashcache has a median 0.62 MB/s and a standard deviation 21.64 MB/s.

When the block size is 16MB, during the one hour period, 70.4% of the total capacity of the SSD is filled, and the SSD hit ratio is 34.4%. On the contrary, when the block size is 4KB, only 0.08% of the total SSD capacity is written and the SSD hit ratio is 0.65%. To make matters worse, 3% of the read requests that hit the SSD miss their deadline due to the poor read throughput. This experiment underscores the importance of using the optimal block size for the SSD to achieve good performance.

5.4.4 Read Throughput Consistency

The important attribute of FlashStream is consistent read throughput from the storage system. Figure 24 shows the variance of storage read throughput for FlashStream and flashcache. The cache is warmed up before the measurement. We run the workload generator for an hour with 50 requests per second for both systems and measure total read throughput of all storage devices (i.e., HDDs and SSDs). A throughput sample is measured with a 1 second window. For FlashStream, the median read throughput is 50.09 MB/s, the standard deviation is 13.68 MB/s, and 50% of throughput samples are in between 40 and 60 MB/s. For flashcache, the median read throughput is 0.62 MB/s and the standard

deviation is 21.64 MB/s. This result shows the superiority of FlashStream for providing consistent storage throughput for read requests.

5.4.5 Energy Efficiency

In this section, we compare the energy efficiency for three different systems: FlashStream and two "traditional" systems (one for more DRAM and another for more disks). Table 6 shows the hardware configuration of the three systems. FlashStream has 2 GB DRAM, 100 GB SSD, and a single 7200 RPM 1 TB HDD. Traditional A has 12 GB DRAM and a single 7200 RPM 1 TB HDD. Traditional B has 2 GB DRAM and two 7200 RPM 1 TB HDDs striped per Linux's software RAID-0 configuration. Compared to FlashStream, Traditional A uses more DRAM for caching instead of SSDs, while Traditional B uses more HDDs for more disk throughput via parallel access. All three systems have the similar total capital cost according to the capital cost of devices in Table 7 (Traditional B is $10 more expensive than the other two). Apache web server is used for traditional systems. Table 7 shows the energy cost of each device in the active state. Each DRAM module in our test machine has 2 GB; 12 GB DRAM corresponds to 6 DIMMs. For FlashStream, we measure the maximum request rate after the SSD is filled up. For traditional systems, we warm up for an hour that is sufficiently long to fill up the given DRAM. We use the same workload as in Section 5.4.1.

Table 6 shows that FlashStream achieves the best energy efficiency. FlashStream serves 64 requests per second with 0% segment miss ratio and its energy efficiency is 2.91 requests per joule. For Traditional configurations, using additional HDD (i.e., Traditional B) is better than using more DRAM (i.e., Traditional A). FlashStream is 94% more energy efficient than Traditional B. Moreover, Traditional B wastes disk capacity because it uses more HDDs only for more throughput; the dataset size is 545 GB while Traditional B has 2 TB HDDs in total. In addition, we ignore the cooling cost and the cost associated with rack space in a datacenter. FlashStream that uses less DRAM and HDDs but more flash-based SSDs would generate less heat and occupy less volume in a rack. We do not consider these benefits in this analysis.

Table 6: Three system configurations for energy efficiency comparison. SSD is a low-end SSD and HDD is 7200 RPM. RPS represents the maximum request rate.

Configuration	DRAM	SSD	HDD	RPS	$\frac{\text{Requests}}{\text{Joule}}$
FlashStream	2GB	100GB	1	64	2.91
Traditional A	12GB	-	1	34	0.57
Traditional B	2GB	-	2	48	1.50

Table 7: Capital cost and energy cost of devices. Data comes from the specification of commodity DRAM, SSDs, and HDDs that are commercially available [13].

	Capital Cost	Energy Cost
DRAM	$8/GB	8W/DIMM
Low-end SSD	$0.8/GB	2W/drive
7200RPM 1TB HDD	$90/drive	12W/drive

5.5 Related Work

In recent years, there have been many different proposals to incorporate flash memory in computer systems. In this section, we summarize prior system studies related to flash memory integration.

Multi-tiered storage system. Flash memory provides faster read latency and consumes less energy than magnetic disks. However, its random writes can be slower than the disks, and it is more expensive than the disks. Enterprise storage systems simultaneously require high performance, low energy consumption, and large capacity. Therefore, tiering flash memory between DRAM and the disks and caching hot data in flash memory are promising ideas to achieve these requirements. Multi-tiered storage systems have been designed in different ways depending on the application workload and system requirements. FlashTier [87] and Flashcache [5] are block device systems; applications and file systems that sit above the block device layer thus automatically reap the benefits of the flash without any modification. Solaris ZFS [19] suggests a multi-tiered storage system in the form of a file system, and applications accessing files on ZFS can get benefit of flash memory without modification. In addition, there have been applications that directly handle DRAM, flash memory, and disks to achieve application specific requirements. FlashStore [43] is a high throughput persistent key-value storage using flash memory as a cache between the RAM and the hard disk. FlashStore is designed for applications that need persistent object store,

and it shows better performance and energy efficiency than BerkeleyDB. Lee et al. [68, 67] have researched on ways to improve database performance with flash memory SSDs. The authors claim that a single enterprise class SSD can be on par with or far better than a dozen hard drives with respect to transaction throughput, cost effectiveness, and energy consumption. Do et al. [44] have proposed three different design alternatives that use an SSD to improve the performance of a database server's buffer manager. They empirically have shown that they could speedup 3-9 times than the default HDD configuration depending on system parameters. Kgil et al. [61] have studied energy efficient web server using flash memory as an extended system memory. Their work is based on simulations with text and image workloads. Singleton et al. [89] have shown how much power can be saved when flash memory is used as a write buffer along with hard disks for mobile multimedia systems.

Flash-only key-value storage system. FAWN [27] is a distributed key-value storage system for low-power and data-internsive computing. FAWN couples low-power embedded CPUs to small amounts of local flash storage, and balances computation and I/O capabilities to enable efficient, massively parallel access to data. Authors of FAWN have shown that FAWN can handle roughly 350 key-value queries per joule, which is two-orders of magnitude more than a disk-based system.

Extended memory. A flash SSD is not used only as fast storage. It can be exploited as slow (but large capacity) memory extending DRAM. FlashVM [86] is a system that proposes using a flash SSD as a dedicated swap device, that provides hints to the SSD for better garbage collection by batching writes, erases, and discards. SSDAlloc [28] is a user-level runtime library that allows developers to treat flash SSDs as an extension of the DRAM in a system. SSDAlloc moves the SSD upward in the memory hierarchy, usable as a larger, slower form of memory instead of a storage alternative for the hard disk. Using these techniques, applications can transparently extend their memory footprint to a few terabytes without any restructuring, far exceeding the DRAM capacities of most servers used in data centers.

The focus of our dissertation research is different from these other research endeavors. FlashStream shows how to construct a high-performance and energy efficient multi-tiered

storage system for adaptive HTTP streaming through the judicious use of flash memory SSDs as an intermediate level in the storage hierarchy. The most important requirement for video streaming is a fast and consistent read latency when accessing video data due to the real-time nature of video streaming. FlashStream is optimized to satisfy the requirement even when used with low-end flash SSDs, and hence, it could achieve both high performance and low energy consumption.

5.6 Summary

In this chapter, we have proposed a set of guidelines for effectively incorporating flash memory SSDs for building a high throughput and power efficient multi-tiered storage system for adaptive HTTP streaming. We have implemented FlashStream based on these guidelines and have evaluated its performance using realistic workloads. We have compared the performance of FlashStream to two other system configurations: Sun's ZFS, and Facebook's flashcache. Similar to FlashStream, the two configurations also incorporate flash memory SSDs in their respective storage hierarchy. FlashStream performs twice as well as its closest competitor, namely, the ZFS configuration, using segment miss ratio as the figure of merit for comparison. In addition, we have compared FlashStream with a traditional two-level storage architecture (DRAM + HDDs), and have shown that, for the same investment cost, FlashStream provides 33% better performance and 94% better energy efficiency.

Until now, we have explored how a multi-tiered storage system using flash memory SSDs can address storage bandwidth problem of CDN servers. In the next chapter, we will explore the networking issues surrounding CDN-based video streaming architecture.

CHAPTER VI

PEER-TO-PEER ADAPTIVE HTTP STREAMING

Peer-to-peer (P2P) streaming technology has emerged as a promising technique for cost-effective and large-scale video streaming on the Internet. Prior studies on P2P streaming generally can be classified into two categories: tree-based overlay multicast [37, 30, 62, 33, 94] and a mesh-based approach [97, 70, 79, 72, 75, 95]. A tree topology is probably the most natural and efficient structure for video streaming, and most of the earlier proposals have adopted the use of tree for overlay multicast. The tree-based systems, however, suffer from node dynamics and thus is subject to frequent reconstruction, which incurs extra cost and renders to sub-optimal structure. On the other hand, a mesh-based approach has been proven to work well on an unreliable network like the Internet and with high peer churn, and several commercial systems such as PPLive [56, 16] and SopCast [17] have been successfully launched based on the mesh-based approach and have attracted millions of users.

In this chapter, we propose a peer-to-peer adaptive HTTP streaming system that integrates the mesh-based P2P streaming technology into the adaptive HTTP streaming architecture. In addition, we evaluate the P2P adaptive HTTP streaming system that has a throughput-smoothing-based adaptation mechanism that is widely used in non-P2P (i.e., client-server) adaptive HTTP streaming systems. We show that the throughput-smoothing-based adaptation mechanism cannot select a proper bitrate and results in poor performance in the P2P architecture. We propose a novel buffer-based adaptation mechanism and compare the performance of the adaptation mechanism for the P2P adaptive HTTP streaming system.

6.1 System Design

Figure 25 illustrates the P2P adaptive HTTP streaming system architecture. In this section, we explain the interactions among the system components.

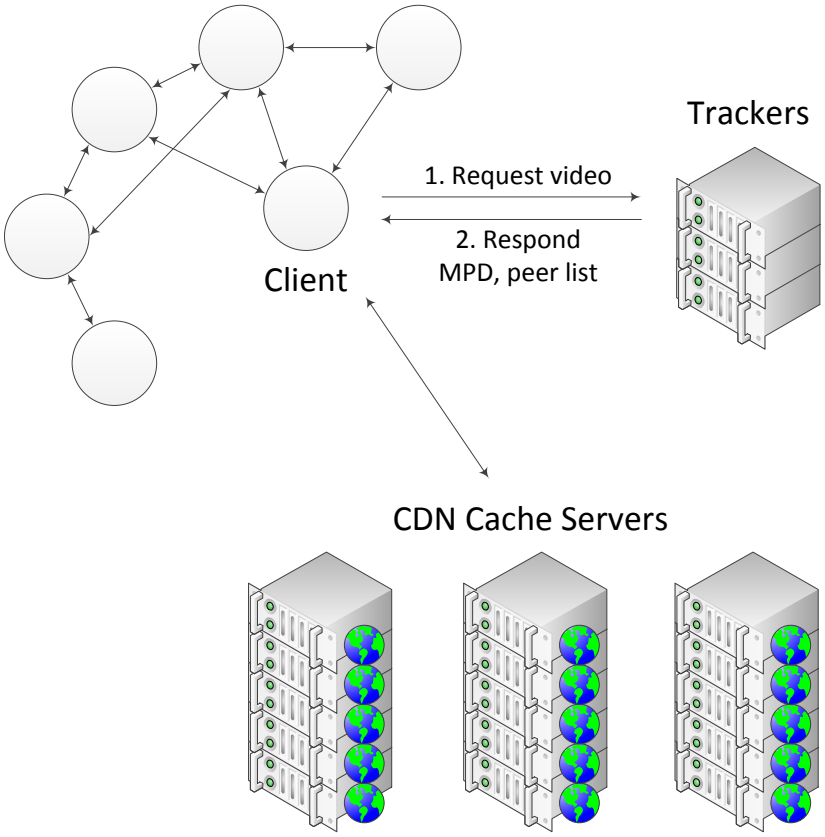


Figure 25: P2P adaptive HTTP streaming system architecture.

6.1.1 Building Blocks

Our P2P adaptive HTTP streaming system has the following major building blocks similar to many mesh-based P2P streaming systems:

- A set of trackers that maintains client information watching a particular video content. Trackers are the first contact point when a client enters into the system. Our current implementation has a single tracker for simplicity, however, it can be easily extended to distributed trackers for scalability by using a distributed hash table (DHT) [52].
- A set of cache servers in the CDN that keeps the source video content.
- A set of client nodes watching the video content. A client selects an appropriate video bitrate during streaming and can download video data from other clients or from

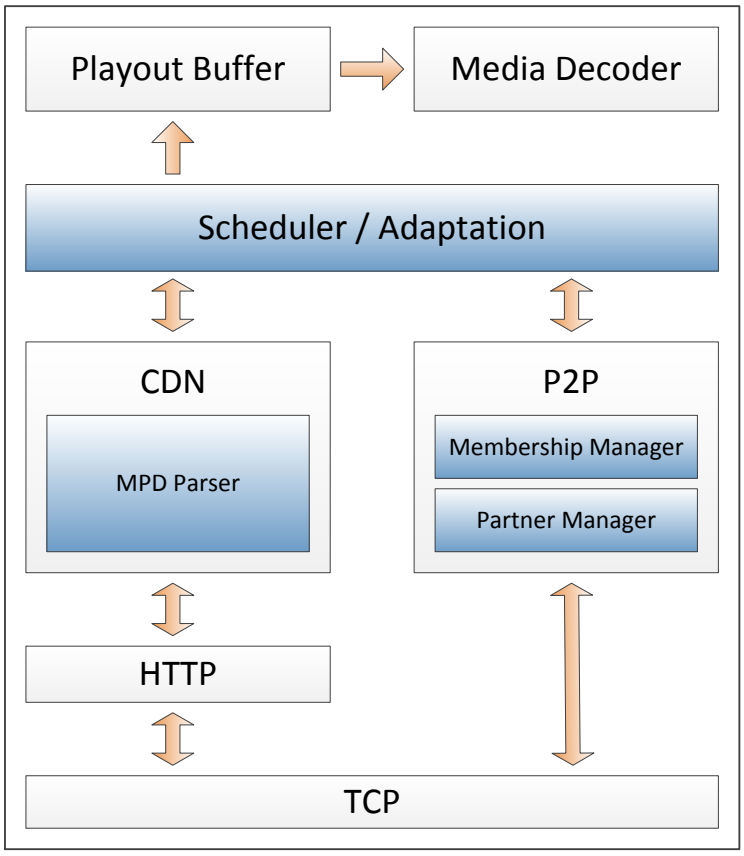


Figure 26: Client node architecture.

CDN cache servers via HTTP. All of our intellectual contributions are contained in this component.

Figure 26 depicts the internal architecture of a client in our P2P adaptive HTTP streaming system. Scheduler decides which segment to fetch and the timing of request for the segment. When requesting a segment, the scheduler selects an appropriate bitrate based on its adaptation algorithm. Our system can parse a manifest file of MPEG-DASH [10], called Media Presentation Description (MPD), which is an XML file that contains information of location (i.e., URL) and timing of video segments. Membership manager and partner manager oversee a P2P overlay network. The scheduler can choose which network to use between the CDN and the P2P network for fetching a segment. A segment can be fetched using only the CDN, only the P2P network, or using both.

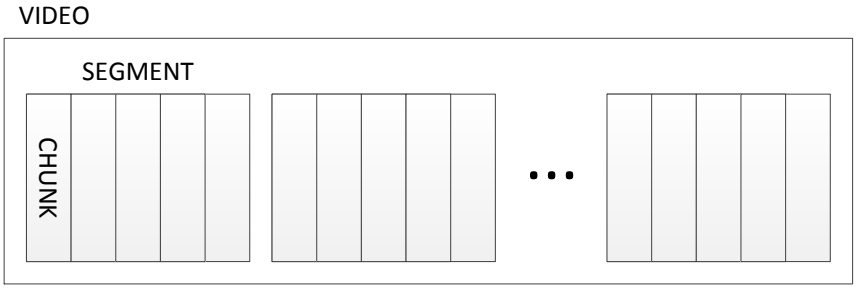


Figure 27: Different units of a media.

6.1.2 Media Segmentation

In the design of a P2P streaming system, a fundamental decision is how to divide a media for the advertisement of media availability and for scheduling the media. In our system, a video is divided into multiple segments, and each segment has the equal length of playback duration similar to existing adaptive HTTP streaming systems. Because a video is encoded in a variable bit rate (VBR), each segment has a different size even though they have the equal playback duration. A segment is a granularity for media availability advertisement. We will take a look at this issue in detail in Section 6.1.4. A segment is further divided into multiple chunks, and a chunk is a granularity for scheduling. We have two options for the chunk size. One is to use a fixed chunk size. The advantage of the fixed chunk size is that we can reduce the message size for scheduling and thus save network bandwidth. When requesting a chunk from a peer, the request message contains only chunk offset because the chunk size is pre-defined. However, when the number of peers that have the segment is larger than the number of chunks of the segment, we use a part of the peers and do not utilize all peers' uplink bandwidth. The second option is to use a variable chunk size. In this case, the chunk size is the segment size divided by the number of peers that have the segment. Because the number of chunks is equal to the number of peers that have the segment, we can achieve maximum parallelism and utilize the full uplink bandwidth of peers. On the other hand, for each chunk request, the request message should contain chunk offset and chunk size. We choose the latter option for our system. We set the minimum chunk size to be 1 KB because too small a chunk size would have a larger header overhead

in transmission and result in inefficient network bandwidth utilization.

6.1.3 Bootstrap

When a client starts to play a video, the client first contacts a tracker server. Each client has a unique identifier (e.g., IP address). The tracker server registers the client identifier to the group of clients watching the same video. The tracker responds to the client with a media presentation description (MPD) file of the requested video and a subset of client identifiers that are randomly selected from the clients in the same video group. The MPD contains information of location (i.e., URL) and timing of segments for a client to fetch and playback the segments of a particular video content. Using the client identifiers returned from the tracker, a client becomes aware of a set of peers who are watching the same video content. Each client maintains membership information that is a partial view of all clients watching the same video. Each client selects a random subset of peer identifiers from its membership information and periodically exchanges the peer identifiers with other peers. We employ Scalable Membership protocol (SCAMP) to distribute membership information among the client nodes. A detailed description of SCAMP can be found in [48]. With the membership gossiping, a client expands the membership information and gets to know more peers who are watching a given video.

6.1.4 Content Discovery

A client node should be able to discover the content they need and which peers are holding the content. The content in cache servers can be identified by the URL in the MPD file downloaded from the tracker when a client joins the system. To figure out the content in peer nodes, we utilize the *buffermap gossiping* mechanism that is widely used in mesh-based P2P streaming systems. Each client node maintains a *buffermap* that is a bitmap indicating whether a particular segment is in its playout buffer. In our system, P2P sharing of segments is confined to segments that are in the playout buffer of the clients; the clients do not store segment files that are outside the playout buffer window. Therefore, we only need a bitmap that is as long as the length of the playout buffer. For example, when the playout buffer contains a maximum of n segments, the buffermap data structure is represented by ($n\times$

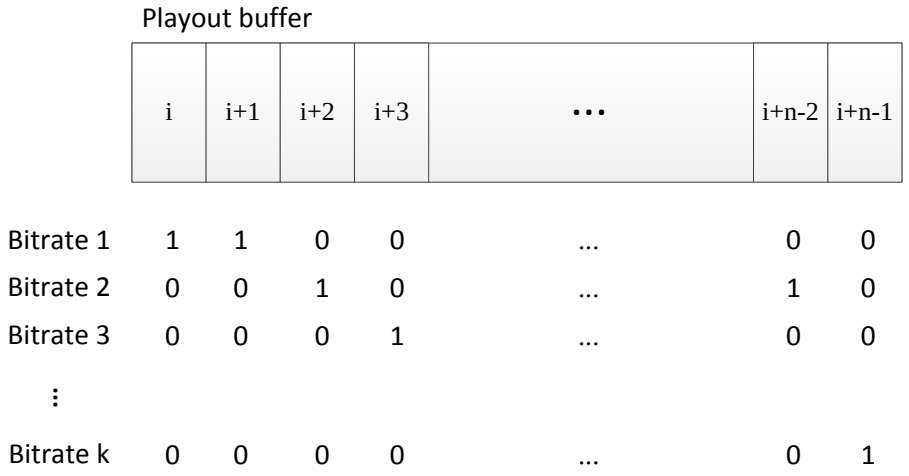


Figure 28: An example of a buffermap. A playout buffer holds n segments at maximum and the first segment in the buffer is i. Segments are encoded in k different bitrates. 1 denotes that a given segment number and a particular bitrate is in the buffer while 0 denotes it is not.

total number of bitrates) bits. Figure 28 illustrates the idea.

A client node selects a random subset of its peers as partners. The node establishes a data stream with each partner, exchanges its buffermap with them periodically, and downloads a chunk of a segment when the partner has the segment in its playout buffer.

6.1.5 Streaming

When a client joins our system for video playback, the client receives an MPD file for the requested video and a partial peer list from a tracker. Then, the client parses the MPD file and starts to send requests of the lowest bitrate segments to CDN cache servers until it fills its playout buffer up to the level of a low watermark (refer to Figure 31). For fast and reliable data transfer at startup, the client uses the lowest bitrate and requests segments from CDN cache servers when filling up the buffer. We call this the *buffering period*. During the buffering period, when a segment is downloaded, then the next segment is requested immediately. In parallel with the buffering, the client exchanges membership information with its peers, selects a subset of them as partners, and exchanges its buffermap with the partners.

Once the buffer is filled up to the level of the low watermark, then the client enters into

a *steady-state period* (or an *ON-OFF period*) [25]. During the steady-state period, segments are requested at the same rate as the playback rate. Therefore, the duration between two segment requests in this period is the same as the playout period of a segment.

When a segment is chosen to be downloaded, a client can download the segment from the CDN cache servers or from its partners. Our system has a configuration parameter, *CDN contribution ratio*, that determines the portion of the segment that is downloaded from the CDN. When the ratio is zero, then the whole segment is downloaded from the P2P network. On the other hand, if the ratio is one, then the whole segment is downloaded from the CDN. When the ratio is in between zero and one, a part of the segment is downloaded from the CDN and the remaining part of the segment is fetched from the P2P network. The size of the CDN part is the CDN configuration ratio times the size of a segment. A part of a segment can be requested from CDN servers using the *byte ranges* protocol of HTTP/1.1 [7]. For the part of the segment that will be downloaded from the P2P network, a client looks up the buffermaps of its partners (that are received during the buffering period) to see if they have the required segment in their respective playout buffer. Then, the P2P part is chopped into chunks (as many as the number of available partners), and the chunks are requested from each available partner. When there are no available partners, then the P2P part of the segment is also fetched from the CDN. Segments are scheduled sequentially both in the buffering period and in the steady-state period.

6.2 Throughput-smoothing-based Adaptation

In the adaptive HTTP streaming paradigm, a video player (i.e., a client) is responsible to efficiently and reliably infer the dynamic network conditions and adapt to the network dynamics for high-quality and smooth video playback. In addition, the inference and the adaptation are required to be conducted above HTTP layer, and this makes the problem more challenging. A conventional adaptation scheme of AHS systems is composed of following steps [71, 59, 73].

1. *Throughput measurement.* AHS systems estimate the current available network bandwidth by dividing the size of the most recent segment requested with the time taken

to fetch the segment.

2. *Throughput smoothing.* The instantaneous segment throughput measured in the previous step is highly variable for each segment and thus unreliable to accurately estimate the network capacity. Using a window of 10 - 20 seconds, the adaptation scheme smooths out the measured throughput samples and uses it as an estimation of the current available network bandwidth. Our implementation uses 20 second smoothing window. Different AHS systems employ different smoothing schemes such as average, 80th-percentile [54], and harmonic-mean [59].

3. *Quantization.* Given the estimated bandwidth, the adaptation scheme then selects a suitable bitrate for the next segment request. Typically, the adaptation scheme selects the maximum available bitrate below the estimated bandwidth.

6.2.1 Evaluation

In this section, we analyze how the throughput-smoothing-based adaptation schemes perform for our P2P adaptive HTTP streaming system. We use a packet-level simulator, OMNeT++ and INET framework [14], for the analysis that can accurately capture the realistic network dynamics and the interactions between TCP and HTTP.

6.2.1.1 Experiment Setting

Big Buck Bunny of DASH dataset [65] is used for our test video data. The segments of the video have 2 second playout length and are encoded in 10 bitrates; 100, 200, 300, 500, 700, 1200, 2000, 3000, 5000, 8000 Kbps.

In our simulation, the physical topology is generated by BRITE [3] using the following configuration parameters: 5 ISPs with 8 routers per ISP in top-down mode (40 routers in total), and 50 end-hosts are evenly attached to the routers. The bandwidth between routers is uniformly distributed between 4 Gbps and 10 Gbps. Each end-host has 1.6 Mbps uplink bandwidth, 6.2 Mbps downlink bandwidth, and 10 ms link delay [63, 91]. The choice of parameters is consistent with a realistic network configuration, wherein the core routers have more bandwidth than the edges [24]. A CDN server is connected to one of the routers

with 100 Gbps uplink/downlink bandwidth and 0.05 μs link delay, which is large enough to serve all clients with the maximum video bitrate; the link of the CDN server is not a bottleneck. Similarly, a tracker server is connected to one of the routers with 100 Gbps uplink/downlink bandwidth and 0.05 μs link delay. Clients join the system according to Poisson arrival with an average inter-arrival time of 1 second. For throughput-smoothing-based adaptation schemes, the playout buffer size is the same as the low watermark (see Figure 31). We have used different low watermarks (e.g., 20 sec, 40 sec, 60 sec, etc), but we only show the results for 40 second low watermark since the results for the other low watermarks show a similar trend. Simulation duration is 10 minutes.

6.2.1.2 Performance Metrics

Measuring video streaming quality is a challenging task because there is no single metric that can measure the streaming quality correctly and different quality metrics are interrelated [29, 74, 46]. For our evaluation, we use three representative metrics that are widely used for comparing performance of adaptive HTTP video streaming systems [59, 71, 29].

- *Buffer underruns:* This is the number of re-buffering events. A re-buffering happens when a video segment that is going to be decoded has not been downloaded in full. We exclude the initial buffering at the start from the calculation. The lower the value, the smoother the video streaming experience.

- *Inefficiency:* Let C be the downlink bandwidth of an end-host, R_t be the video bitrate at time t, and T be the duration of the measurement. The inefficiency is defined as $\frac{\sum_{t=1}^{T} |C - R_t|}{C \cdot T}$. The lower the value, the more efficiently the network bandwidth is utilized to deliver better picture quality to a client.

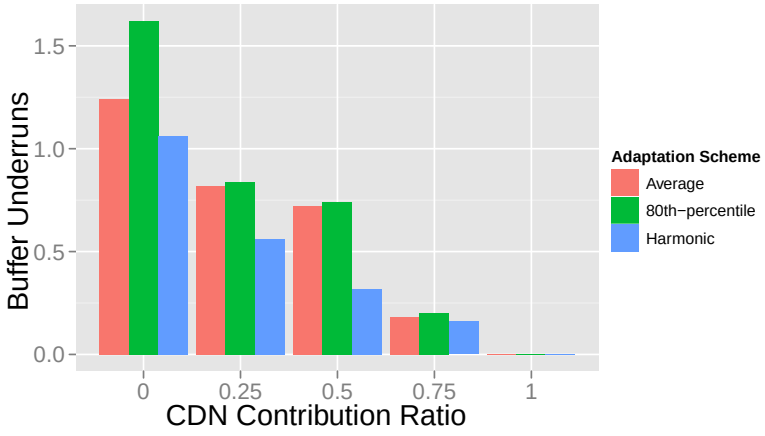
- *Instability:* User studies suggest that users do not tolerate too frequent bitrate switches as it impacts their perceptual experience [40]. The instability metric is defined as $\sum_{t=k+1}^{T} \frac{\sum_{d=0}^{k-1} |R_{t-d} - R_{t-d-1}| \cdot w(d)}{(T-k) \sum_{d=1}^{k} R_{t-d} \cdot w(d)}$, where $w(d) = k - d$ is a weight function that puts more weight on more recent samples. $k = 20$ is used for our calculation.

These metrics are computed for each client. In the following sections, we show the average value of the metrics computed using a total of 50 clients.

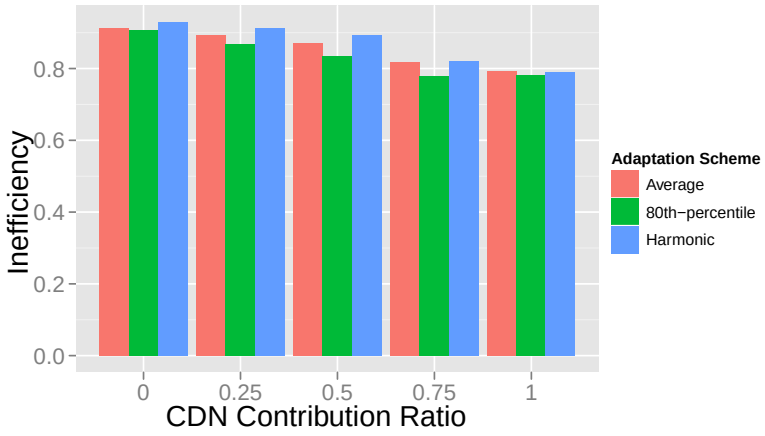
6.2.1.3 Analysis

Figure 29(a) shows the number of buffer underruns that clients experience. The CDN contribution ratio is the portion of a video segment that is downloaded from the CDN. From the graph, for all adaptation schemes, we can see that the buffer underrun happens less as a client utilizes CDN more when fetching a segment. When a client relies on only P2P network for fetching a segment, it experiences at least 1 buffer underrun for all the adaptation schemes. From Figure 29(b), inefficiency also decreases as CDN is utilized more. Less inefficiency means that the clients choose higher bitrate segments and utilize their downlink bandwidth (i.e., 6.2 Mbps) more. The 80th-percentile scheme tends to choose higher bitrate more aggressively than the other schemes. Therefore, the 80th-percentile scheme has lower inefficiency; but, adversely, it has higher buffer underruns and instability compared to the other schemes. Figure 29(c) shows that, interestingly, the instability is higher when chunks are downloaded from both CDN and P2P than when they are fetched only from CDN or only from P2P. Overall, throughput-smoothing-based adaptation schemes need larger CDN contribution for smoother streaming with a better picture quality.

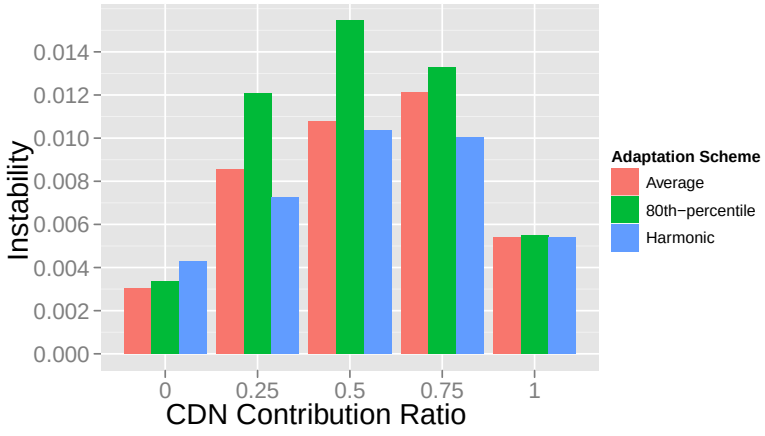
In a client-server adaptive HTTP streaming architecture, the assumption behind the adaptive scheme is that the uplink bandwidth of servers is not a bottleneck; the downlink of the client is the only bottleneck. Therefore, the throughput-smoothing-based adaptive scheme uses the segment download throughput as a cue for estimating the network capacity. However, this is not true in a P2P architecture. In the P2P architecture, the clients as well as the servers can serve as sources for video segments. In addition, the uplink bandwidth is far smaller than the downlink bandwidth in a residential network [63, 91]. Even worse, the mesh-based P2P approach does not dedicate a node's uplink bandwidth to a particular partner. Any partner can request data at any point of time, therefore, the number of active data streams that share the narrow uplink bandwidth of a node dynamically varies over time. Owing to such reasoning, our hypothesis is that the throughput-smoothing-based



(a) Buffer Underruns

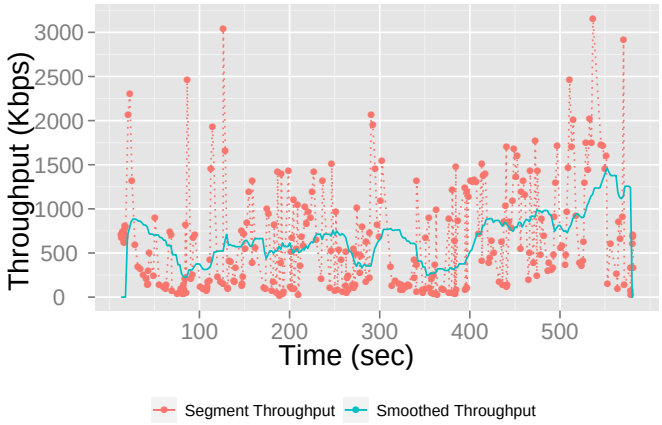


(b) Inefficiency

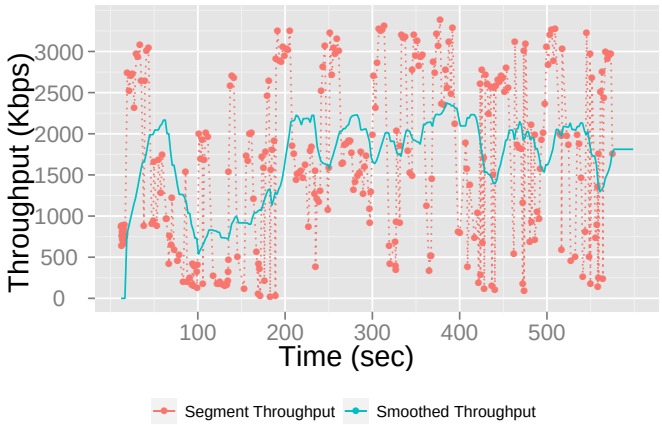


(c) Instability

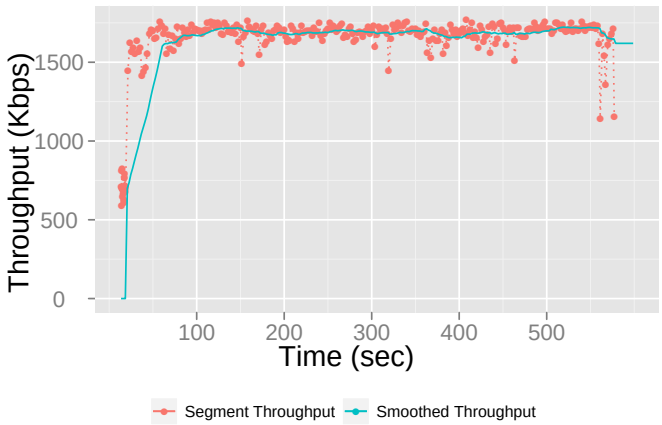
Figure 29: Performance of throughput-smoothing-based adaptation schemes against different CDN contribution ratios. Playout buffer size is 40 seconds.



(a) CDN Contribution Ratio 0.0



(b) CDN Contribution Ratio 0.5



(c) CDN Contribution Ratio 1.0

Figure 30: Segment throughput significantly oscillates when P2P network is used for fetching segments. On the other hand, segment throughput is very stable when only CDN is used for fetching segments. Average smoothing scheme is used. The smoothed throughput cannot correctly infer the available network capacity when P2P network is used.

adaptation cannot estimate a client's network capacity correctly in a P2P architecture.

Figure 30 shows a client's segment throughput and its smoothed value during a 10 minute video playback for different CDN contribution ratios. When P2P network is used for fetching segments, the segment throughput of a client node significantly oscillates due to the relentless P2P dynamics. Therefore, the smoothed value of the throughput cannot be a proper cue for estimating the available network bandwidth of the client node. On the other hand, in the case of CDN only, the segment throughput of a client is very stable. As can be seen from Figure 29(a), the system experiences no buffer underruns when the CDN contribution ratio is 1.0 (i.e., clients fetch segments only from the CDN). This result confirms our hypothesis that the throughput-smoothing-based technique for estimating the available network bandwidth at a client is only appropriate for a CDN-only architecture and not for a hybrid CDN-P2P architecture.

6.3 Buffer-based Adaptation

In the previous section, we observed that the throughput-smoothing-based adaptation scheme cannot reliably estimate the available network bandwidth in the P2P architecture because of the P2P dynamics and narrow uplink bandwidth of a client node. Then, what is the solution for this problem? Before answering this question, we should answer the following questions step by step.

- What is the purpose of measuring the segment throughput? We use the segment throughput for estimating the available network capacity.
- Then, what is the purpose of estimating the network capacity? We select a video bitrate smaller than the network capacity.
- Why do you select a video bitrate smaller than the estimated network capacity? It is to *avoid buffer underrun*.

The ultimate reason for the throughput-smoothing is to avoid the buffer underrun for smooth playback without interrupts. Therefore, our idea is to avoid bandwidth estimation altogether and focus on the playout buffer occupancy, and control the video bitrate so as not

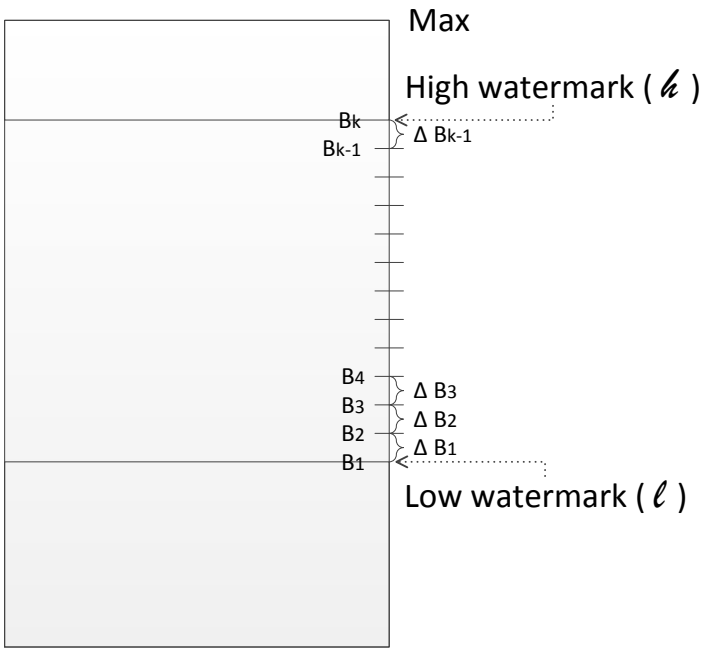


Figure 31: Playout buffer is divided into k levels where k is the number of bitrates encoded for a video. Whenever the current buffer level crosses a marked buffer level, then the adaptation scheme changes the bitrate for the segment that will be fetched next.

to empty the playout buffer. Figure 31 depicts our idea. Suppose that a video is encoded in k bitrates $(R_1, R_2, ..., R_k)$ where R_1 is the minimum bitrate and R_k is the maximum bitrate. To choose a bitrate based on the buffer occupancy, we need a mapping from a buffer level to a bitrate. Low watermark (l) is mapped to the minimum bitrate (R_1), while high watermark (h) is mapped to the maximum bitrate (R_k). When the buffer level goes down under the low watermark, then the adaptation scheme selects only the lowest bitrate, R_1. On the other hand, if the buffer level goes up beyond the high watermark, the the adaptation scheme selects the highest bitrate, R_k. Between the low watermark and high watermark, the buffer is divided into k buffer levels. We define a buffer level, $B_i = l + \sum_{j=1}^{i-1} \Delta B_j$, where $B_1 = l, B_k = h$, and B_i is mapped to R_i. Let's suppose the current buffer level is denoted as B_{cur}. If $B_i \leq B_{cur} < B_j$, then the adaptation scheme selects a bitrate R_i. Therefore, whenever the current buffer level crosses a marked buffer level (B_i), then the adaptation scheme changes the bitrate for the segment that will be fetched next. The

variables, l and ΔB_i, are configuration parameters that a system administrator can choose. There are trade-offs associated with the choice of these parameters. The buffer below the low watermark is filled with the lowest bitrate segments, therefore, a client would end up watching a lower picture quality for a longer duration when the low watermark is higher. When ΔB_i becomes larger, the adaptation scheme will change bitrates less frequently, but it requires more buffer capacity.

Adaptive HTTP streaming systems that use the throughput-smoothing-based adaptation scheme goes into the ON-OFF period and fetches segments periodically when the buffer is filled up to the level of low watermark. On the other hand, our system that uses the buffer-based adaptation scheme continues fetching segments *immediately*, limited by the maximum buffer size, even though the buffer is filled up beyond the low watermark. Therefore, our scheme requires a larger playout buffer than the throughput-smoothing-based schemes.

6.3.1 Evaluation

We use the same evaluation setup as in Section 6.2.1. For the buffer-based adaptation scheme, we set the high watermark (h) to be 160 seconds and the max buffer size to be 170 seconds. We let the buffer levels (B_i) be equally spaced between the low watermark (l) and the high watermark (h); $\Delta B_i = \frac{h-l}{k}$. The CDN contribution ratio and the low watermark are the control variables. In this section, we compare performance of our buffer-based adaptation scheme with throughput-smoothing-based schemes for different settings of the control variables. Same as Section 6.2.1, we use buffer underruns, inefficiency, and instability as performance metrics.

6.3.1.1 Analysis

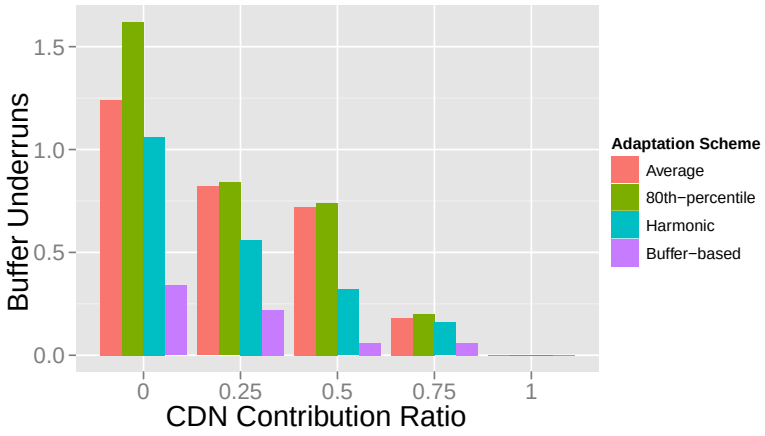
Figure 32 shows the performance of four adaptation schemes for different CDN contribution ratios with a fixed low watermark (i.e., 40 seconds). Clients experience much less buffer underruns with the buffer-based scheme compared to the throughput-smoothing-based schemes, especially, for smaller CDN contribution ratio. Inefficiency of the buffer-based scheme is larger than throughput-smoothing-based schemes when the CDN contribution ratio is less than or equal to 0.5. This result points to the fact that the buffer-based

scheme selects lower video bitrate, and consequently utilizes less of the available downlink bandwidth. However, on the positive side, the buffer-based scheme experiences less buffer underruns due to this conservative choice of lower bitrate. As we mentioned earlier, frequent buffer underruns leading to "re-buffering" has been found to be the most detrimental factor for user experience in video streaming [46]. When the P2P network is used (CDN contribution ratio is less than 1), then the buffer-based scheme shows the lowest instability. On the other hand, when segments are fetched only from CDN, the segment throughput is very stable (Figure 30(c)), and hence throughput-smoothing-schemes show lower instability than the buffer-based scheme.

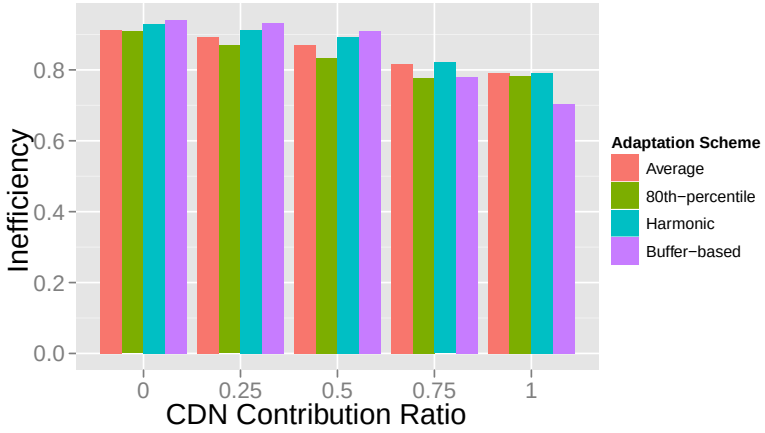
Following conclusions can be drawn from the results:

- Buffer-based scheme avoids buffer underruns better than throughput-smoothing-based schemes for all CDN contribution ratio settings.
- Buffer-based scheme is careful in not choosing unnecessarily high bitrates (based on transient network conditions) that could lead to frequent buffer underruns and the system entering into the "re-buffering" state.
- Buffer-based scheme shows lower instability than throughput-smoothing-based schemes when the P2P network is used in conjunction with the CDN.

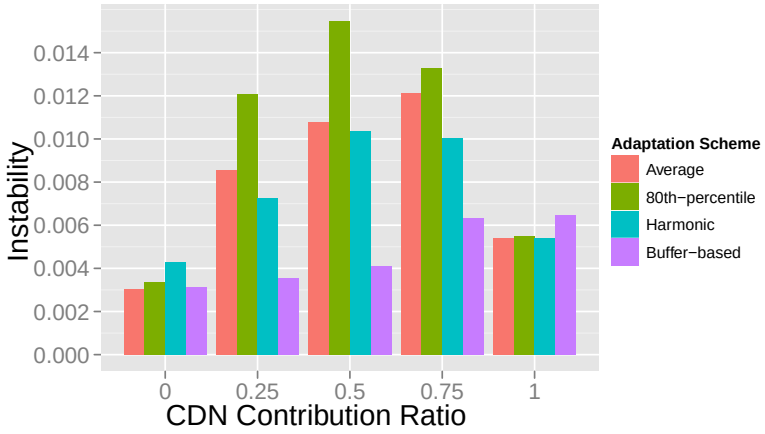
Next, we compare the performance of the adaptation schemes for different low watermarks with the CDN contribution ratio set to 0. A smaller low watermark has a higher chance of depleting the playout buffer. Figure 33(a) shows that with a smaller low watermark, all schemes tend to have higher buffer underruns. The buffer-based scheme has the least buffer underruns compared to throughput-smoothing-based schemes at all low watermarks. We notice from Figure 33(b) that the low watermark level does not affect the choice of bitrate. Harmonic-mean and buffer-based adaptations select lower bitrate (thus, higher inefficiency) than the other schemes, but they have less buffer underruns by not choosing unnecessarily high bitrate. Figure 33(c) shows that the instability is not sensitive to the low watermark level for values larger than 20 seconds. When the low watermark is 20 seconds, the throughput-smoothing-based schemes have a high chance of buffer underruns



(a) Buffer Underruns

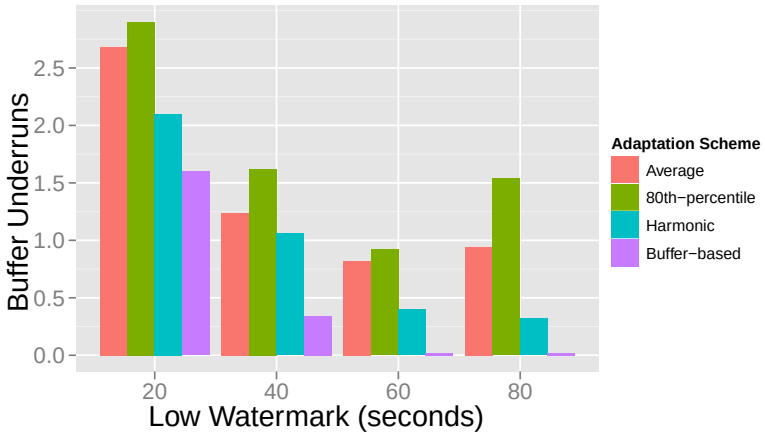


(b) Inefficiency

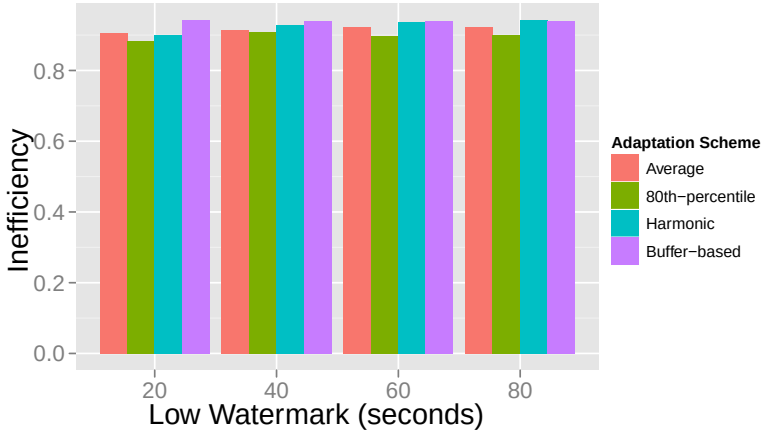


(c) Instability

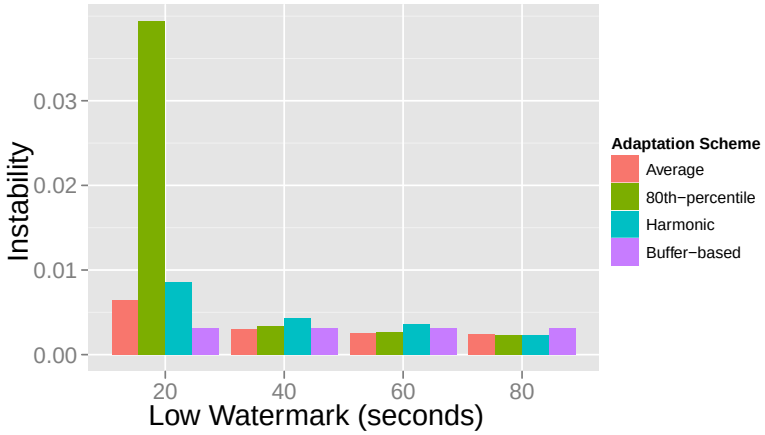
Figure 32: Performance of adaptation schemes against different CDN contribution ratios. Low watermark is 40 seconds.



(a) Buffer Underruns



(b) Inefficiency



(c) Instability

Figure 33: Performance of adaptation schemes against different low watermarks. CDN contribution ratio is 0.

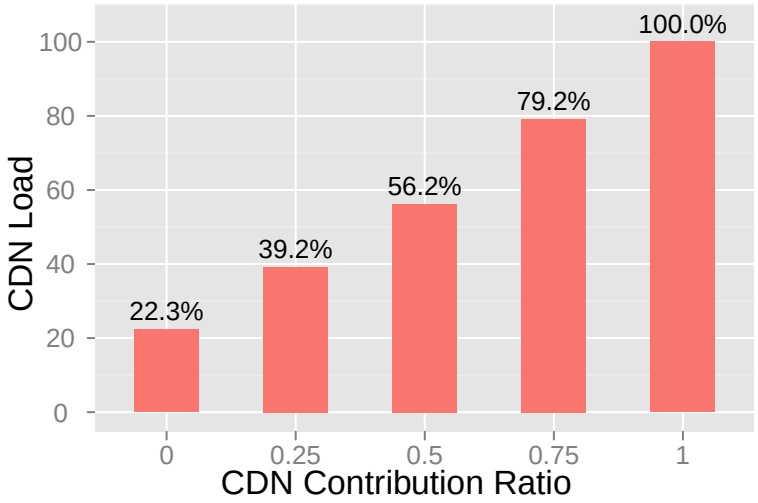


Figure 34: Buffer-based adaptation scheme with 40 second low watermark is used. The P2P network can serve 77.7% of total video traffic while CDN serves only 22.3% of the total when the CDN contribution ratio is 0.

forcing the clients to frequently switch to the buffering state and download the lowest bitrate segments. This is the reason for the increased instability of throughput-smoothing-based schemes when the low watermark value is set to 20 seconds

Following conclusions can be drawn from the results:

- Buffer-based scheme avoids buffer underruns better than the throughput-smoothing-based schemes for all settings of the low watermarks.
- Buffer-based scheme does not choose unnecessarily high bitrate that can cause buffer underruns.
- Buffer-based scheme shows lower instability than the throughput-smoothing-based schemes when the low watermark setting is less than or equal to 60 seconds.

Finally, we study the impact of the P2P network in reducing the CDN server load. Figure 34 shows the traffic served by CDN against different CDN contribution ratios when the low watermark is 40 seconds and the buffer-based scheme is used. When the CDN contribution ratio is 0, CDN serves only the segments that are not available from the P2P network and/or the segments fetched in the buffering period. In this case, the P2P network can serve 77.7% of total video traffic thus reducing the CDN server load to 22.3%.

Even with such a small CDN contribution, our buffer-based scheme shows very low buffer underruns compared to the other schemes and thus provides smooth video streaming (see Figure 32(a)). With a larger CDN contribution ratio, more traffic is served by the CDN.

In this section, we have explored how different aspects of video streaming quality (i.e., buffer underruns, inefficiency, instability) are affected by the control parameters (CDN contribution ratio and low watermark setting) for the different client-side adaptation schemes. We have shown that the buffer-based scheme does not choose unnecessarily high bitrate (therefore, a larger inefficiency in some cases) and avoids buffer underruns much better than throughput-smoothing-based schemes for all settings of the control parameters. In addition, the buffer-based scheme changes bitrate less frequently and more smoothly than throughput-smoothing-based schemes when the P2P network is utilized.

Our performance results can also serve a "prescriptive role" for service providers to tune their system. For example, if the service provider wishes to have the following service requirement:

- buffer underrun less than 0.4,
- inefficiency less than 0.95, and
- instability less than 0.1

When the low watermark is set to 40 seconds, these service requirements can be met with the following settings (see Figure 32):

- buffer-based adaptation scheme with the CDN contribution ratio set to 0; or
- Harmonic-mean scheme with the CDN contribution ratio set to 0.75.

It should be noted that the buffer-based scheme can achieve this policy requirement with much less network traffic to the CDN servers.

On the other hand, if the service requirement calls for:

- buffer underruns less than 0.01,
- inefficiency less than 0.8, and

- instability less than 0.1.

These service requirements can be met by any of the schemes (see Figure 32) with the following settings:

- CDN contribution ratio set to 1, and
- low watermark set to 40 seconds.

It should be noted that the buffer-based scheme achieves the lowest inefficiency in meeting these requirements (and hence the best picture quality).

6.4 Related Work

Adaptive HTTP Streaming (AHS) is a new video streaming paradigm on the Internet. Rather than rely on the dedicated video servers of yesteryears, AHS exploits off-the-shelf web servers for video streaming. Various qualitative and quantitative studies [26, 42, 59, 76] have examined the performance of AHS streaming in commercial systems such as Netflix, Microsoft's Smooth Streaming, Apple's HTTP Live Streaming, and Adobe's Dynamic Streaming. Through experiments in controlled settings, these studies characterize the behavior of AHS streaming systems when one or more video players share a bottleneck link to the video server.

Many research studies have proposed quality adaptation schemes for AHS streaming [59, 71, 73, 76]. These systems analyze metrics such as fairness, efficiency, stability, etc. The majority of the proposed schemes employ a throughput-smoothing-based adaptation mechanism for AHS streaming. Other systems have proposed control-theoretic approaches and Markov-Decision Process techniques for improving the overall performance of the bitrate adaptation [92, 98]. Similar to our buffer-based adaptation scheme, Huang et al. [55] have proposed an adaptation scheme that is based on the playout buffer occupancy. They have proposed a similar idea with a different motivation from ours. The authors argue that when multiple AHS clients in a client-server architecture compete each other for a bottleneck link, the traditional throughput-smoothing-based schemes estimate the network capacity inaccurately, thus choose inappropriate bitrate. However, they have not compared

performance of their scheme with throughput-smoothing-based schemes, and they have not shown the superiority of their scheme for P2P adaptive HTTP streaming.

We have proposed a peer-to-peer adaptive HTTP streaming system that integrates the mesh-based P2P streaming technology into the adaptive HTTP streaming architecture. We have shown that the throughput-smoothing-based adaptation scheme does not work well in the P2P architecture and proposed a novel buffer-based adaptation scheme for a solution.

6.5 Summary

In this chapter, we have illustrated the design of our P2P adaptive HTTP streaming system that combines a mesh-based P2P streaming system and an adaptive HTTP streaming system. We have shown that throughput-smoothing-based adaptation schemes that are widely used for client-server adaptive HTTP streaming systems have unnecessary buffer underruns in the P2P environment. To address the problem, we have proposed a buffer-based adaptation scheme that only focuses on the playout buffer occupancy and selects bitrates accordingly. Our buffer-based scheme is robust to the P2P dynamics and does not have unnecessary buffer underruns. We have shown that our P2P adaptive HTTP streaming system with the buffer-based adaptation scheme has a great potential for saving CDN traffic depending on the streaming quality requirements of a video streaming service provider.

CHAPTER VII

CONCLUSION AND FUTURE WORK

Need for high-bitrate on-demand video content on the Internet is exploding. To satisfy reliable on-demand video streaming, a video service provider should provide both a significant amount of storage bandwidth and network bandwidth. Support for such a huge amount of bandwidth requires a significant cost burden to the service provider. Such a cost burden is aggravated in the adaptive HTTP streaming (AHS) paradigm because the AHS systems rely on overprovisioned content distribution network (CDN) resources for a reliable video delivery over the Internet.

In this dissertation, we first show that hard disk drives (HDDs) are not ideal for serving video content in the AHS systems; The video segment size could be very small and diverse in the AHS systems, and HDDs are very slow for reading such small files. On the other hand, flash memory solid-state drives (SSDs) are very fast for reading small files and consume a lot less energy than HDDs. However, flash-based SSDs has smaller capacity than HDDs for a given cost. Therefore, a multi-tiered storage where flash-based SSDs sit in the middle between DRAM and HDDs has a great potential for adaptive HTTP streaming that needs large capacity and large bandwidth at the same time. Next, we evaluate our hypothesis using state-of-the-art multi-tiered storage systems with AHS workloads. Different from our expectation, such systems have shown less or no performance advantage compared to traditional two-tier storage systems for adaptive HTTP streaming. We analyze the reasons for the counter-intuitive results, suggest design guidelines for constructing a multi-tiered storage system for adaptive HTTP streaming, and propose our FlashStream system that is designed and implemented based on the guidelines. For the same storage cost, FlashStream provides 33% better performance and 94% better energy efficiency compared to the two-tier storage architecture. Finally, we propose a P2P adaptive HTTP streaming system with a novel buffer-based adaptation scheme that provides reliable video delivery while significantly

reducing the traffic going to the servers. With the FlashStream multi-tiered storage system and the P2P AHS system, we could maximize AHS system performance for a given cost.

There are several avenues for future research building on our study for a multi-tiered storage system. NAND Flash memory is the first non-volatile memory technology that is currently widely used in computer systems. In recent years, there have been many different proposals on ways to incorporate flash memory in computer systems. Flash memory has impacted greatly both embedded systems and high-performance systems. Our own studies and related work help surface research issues that new non-volatile memories will bring to the forefront both from the point of view of the computer architecture and system software structure. With the advent of *storage class memories (SCM)* the distinction between primary (physical) memory and storage is getting blurred. NAND Flash memories have been around for a while and they have been used inside Solid State Drives (SSDs) quite successfully as a viable alternative to the hard disk drive. However, NAND Flash has serious performance and reliability problems (e.g., poor random write, wear-out issue). While NAND Flash is also a type of SCM, the exciting change in recent times is the advent of newer technologies such as *phase change memories (PCM)* [88], and *spin torque transfer (STT)* magnetic RAM [38]. With improvements in process technology for making PCMs, it is expected to scale much better than NAND Flash, well into the single digit nanometer feature sizes. PCM has read and write latencies that are about two orders of magnitude lower than NAND Flash (i.e., closer to DRAM latencies). PCM also has better durability than NAND Flash: few tens of millions writes per cell, as opposed to only a few thousand writes for NAND Flash. Perhaps what makes PCM so attractive is the fact that it can be used to design byte-addressable persistent memory, making them a viable competitor to DRAM. At this point of time, PCM is the most mature of the newer SCM technologies. However, STTRAM has low access latencies (< 20 ns) and is even more promising as a future competitor to DRAM. Perhaps what is more exciting is the fact that computer architects have been eyeing these technologies from the point of view of replacing the DRAM with such non-volatile counterparts [83, 66, 34]. Until recently, for over 50 years, system research have had assumptions of two-level memory and storage; fast volatile memory and

slow non-volatile storage devices. Although the performance of storage increased, but the gap between memory and storage seldom diminished before advent of NAND Flash based SSDs. But, what if those assumptions are no more valid, so it is possible to reveal a fast non-volatile memory instead of two separate memory hierarchy structures? Instead of using non-volatile memory as a fast storage device which replaces mechanical HDD drives, how can we use new persistent memory in more revolutionary ways? This change more than any other aspect of the computer organization has the potential of requiring a revolutionary approach to the way system software is built, in particular, the operating system itself. The entire system software stack should be re-designed including virtual memory, file systems, process schedulers, and even protection mechanisms.

REFERENCES

[1] "Apache." http://httpd.apache.org.

[2] "Blktrace." http://linux.die.net/man/8/blktrace.

[3] "BRITE." http://www.cs.bu.edu/brite/.

[4] "Ext4 file system." https://ext4.wiki.kernel.org.

[5] "Flashcache." http://www.facebook.com/note.php?note_id=388112370932.

[6] "Hdd technology trends." http://www.storagenewsletter.com/news/disk/hdd-technology-trends-ibm.

[7] "Http/1.1." http://www.ietf.org/rfc/rfc2616.txt.

[8] "Hulu." http://www.hulu.com.

[9] "Hulu viewers." http://www.comscore.com/Press_Events/Press_Releases/2011/12/comScore_Releases_November_2011_U.S._Online_Video_Rankings.

[10] "ISO/IEC DIS 23009-1.2." Information technology - Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats.

[11] "Netflix." http://www.netflix.com.

[12] "Netflix traffic." http://www.techspot.com/news/46048-netflix-represents-327-of-north-americas-peak-web-traffic.html.

[13] "Newegg." http://www.newegg.com.

[14] "OMNeT++." http://www.omnetpp.org/.

[15] "pion-net." http://www.pion.org/projects/pion-network-library.

[16] "PPLive." http://www.pplive.com.

[17] "SopCast." http://www.sopcast.org.

[18] "YouTube." http://www.youtube.com.

[19] "Zettabyte file system." http://solaris-training.com/classp/200_HTML/docs/zfs_wp.pdf.

[20] "ZFS L2ARC." https://blogs.oracle.com/brendan/entry/test.

[21] Acharya, S. and Smith, B., "Characterizing user access to videos on the world wide web," in *In Proceedings of MMCN*, 2000.

[22] Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L., "Unreeling netflix: Understanding and improving multi-cdn movie delivery.," in *INFOCOM*, pp. 1620–1628, IEEE, 2012.

[23] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R., "Design tradeoffs for ssd performance," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, (Berkeley, CA, USA), pp. 57–70, USENIX Association, 2008.

[24] Akella, A., Seshan, S., and Shaikh, A., "An empirical evaluation of wide-area internet bottlenecks," in *Proceedings of IMC*, 2003.

[25] Akhshabi, S., Anantakrishnan, L., Begen, A. C., and Dovrolis, C., "What happens when http adaptive streaming players compete for bandwidth?," in *Proceedings of the 22nd International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '12, pp. 9–14, 2012.

[26] Akhshabi, S., Begen, A. C., and Dovrolis, C., "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pp. 157–168, 2011.

[27] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V., "Fawn: a fast array of wimpy nodes.," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, (Big Sky, MT, USA), ACM, October 2009.

[28] Badam, A. and Pai, V. S., "Ssdalloc: hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, March 2011.

[29] Balachandran, A., Sekar, V., Akella, A., Seshan, S., Stoica, I., and Zhang, H., "Developing a predictive model of quality of experience for internet video," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pp. 339–350, 2013.

[30] Banerjee, S., Bhattacharjee, B., and Kommareddy, C., "Scalable application layer multicast," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, 2002.

[31] Barbir, A., Cain, B., Nair, R., and Spatscheck, O., "Known content network (cn) request-routing mechanisms." RFC 3568, `http://tools.ietf.org/html/rfc3568`.

[32] Begen, A. C., Akgul, T., and Baugher, M., "Watching video over the web: Part1: Streaming protocols," *IEEE Internet Computing*, vol. 15, no. 2, pp. 54–63, 2011.

[33] Castro, M., Druschel, P., Kermarrec, A. M., Nandi, A., Rowstron, A., and Singh, A., "Splitstream: High-bandwidth multicast in cooperative environments," in *Proceedings of ACM SOSP*, (New York, USA), October 2003.

[34] Caulfield, A. M., Coburn, J., Mollov, T., De, A., Akel, A., He, J., Jagatheesan, A., Gupta, R. K., Snavely, A., and Swanson, S., "Understanding

the impact of emerging non-volatile memories on high- performance, io-intensive computing," in *In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pp. 1–11, IEEE Computer Society, Washington, DC, USA, 2010.

[35] Cha, M., Kwak, H., Rodriguez, P., Ahn, Y.-Y., and Moon, S., "I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pp. 1–14, 2007.

[36] Chervenak, A. L., "Tertiary storage: An evaluation of new applications," *Ph.D. Thesis, University of California at Berkeley, Computer Science Division Technical Report UDB/CSD 94/847*, December 1994.

[37] Chu, Y. H., Rao, S. G., and Zhang, H., "A case for end system multicast," in *Proceedings of ACM SIGMETRICS*, June 2000.

[38] Chung, S., Rho, K.-M., Kim, S.-D., Suh, H.-J., Kim, D.-J., Kim, H.-J., Lee, S.-H., Park, J.-H., Hwang, H.-M., Hwang, S.-M., Lee, J.-Y., An, Y.-B., Yi, J.-U., Seo, Y.-H., Jung, D.-H., Lee, M.-S., Cho, S.-H., Kim, J.-N., Park, G.-J., Jin, G., Driskill-Smith, A., Nikitin, V., Ong, A., Tang, X., Kim, Y., Rho, J.-S., Park, S.-K., Chung, S.-W., Jeong, J.-G., and Hong, S.-J., "Fully integrated 54nm STT-RAM with the smallest bit cell dimension for high density memory application," in *Electron Devices Meeting (IEDM), 2010 IEEE International*, pp. 12.7.1 –12.7.4, December 2010.

[39] Cisco, "Cisco Visual Networking Index: Forecast and Methodology, 2012 - 2017." `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html`.

[40] Cranley, N., Perry, P., and Murphy, L., "User perception of adapting video quality," *Int. J. Hum.-Comput. Stud.*, vol. 64, pp. 637–647, August 2006.

[41] Dan, A., Sitaram, D., and Shahabuddin, P., "Scheduling policies for an on-demand video server with batching," in *in Proc. of ACM Multimedia*, pp. 15–23, 1994.

[42] De Cicco, L. and Mascolo, S., "An experimental investigation of the akamai adaptive video streaming," in *Proceedings of the 6th International Conference on HCI in Work and Learning, Life and Leisure: Workgroup Human-computer Interaction and Usability Engineering*, USAB'10, pp. 447–464, 2010.

[43] Debnath, B., Sengupta, S., and Li, J., "Flashstore: High throughput persistent key-value store," in *Proceedings of the 36th International Conference on Very Large Data Bases*, (Singapore), September 2010.

[44] Do, J., Zhang, D., Patel, J. M., DeWitt, D. J., Naughton, J. F., and Halverson, A., "Turbocharging dbms buffer pool using ssds," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pp. 1113–1124, 2011.

[45] Dobrian, F., Sekar, V., Awan, A., Stoica, I., Joseph, D., Ganjam, A., Zhan, J., and Zhang, H., "Understanding the impact of video quality on user engagement,"

in *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pp. 362–373, 2011.

[46] Dobrian, F., Sekar, V., Awan, A., Stoica, I., Joseph, D., Ganjam, A., Zhan, J., and Zhang, H., "Understanding the impact of video quality on user engagement," in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pp. 362–373, 2011.

[47] Finamore, A., Mellia, M., Munafò, M. M., Torres, R., and Rao, S. G., "Youtube everywhere: impact of device and infrastructure synergies on user experience," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pp. 345–360, 2011.

[48] Ganesh, A. J., Kermarrec, A.-M., and Massouli, L., "Peer-to-peer membership management for gossip-based protocols," *Computers, IEEE Transactions on*, vol. 52, no. 2, pp. 139–149, 2003.

[49] Gemmell, D. J. and Han, J., "Multimedia network file servers: multichannel delay-sensitive data retrieval," *Multimedia Syst.*, vol. 1, pp. 240–252, April 1994.

[50] Gemmell, J., Vin, H. M., Kandlur, D. D., Rangan, P. V., and Rowe, L. A., "Multimedia storage servers: A tutorial," *Computer*, vol. 28, no. 5, pp. 40–49, 1995.

[51] Gray, J. and Fitzgerald, B., "Flash disk opportunity for server-applications," *http://www.research.microsoft.com/~gray*, January 2007.

[52] Gupta, A., Liskov, B., and Rodrigues, R., "One hop lookups for peer-to-peer overlays," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pp. 2–2, 2003.

[53] Huang, C., Li, J., and Ross, K. W., "Can internet video-on-demand be profitable?," in *Proceedings of the ACM SIGCOMM 2007 conference*, SIGCOMM '07, pp. 133–144, 2007.

[54] Huang, T.-Y., Handigol, N., Heller, B., McKeown, N., and Johari, R., "Confused, timid, and unstable: Picking a video streaming rate is hard," in *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pp. 225–238, 2012.

[55] Huang, T.-Y., Johari, R., and McKeown, N., "Downton abbey without the hiccups: Buffer-based rate adaptation for http video streaming," in *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, FhMN '13, pp. 9–14, 2013.

[56] Huang, Y., Fu, T., Chiu, D., Lui, J., and Huang, C., "Challenges, design and analysis of a large-scale p2p-vod system," in *Proceedings of ACM SIGCOMM*, (Seattle, WA, USA), 2008.

[57] Hwang, C.-G., "Nanotechnology enables a new memory growth model," in *Proceedings of the IEEE 91(11)*, pp. 1765–1771, November 2003.

[58] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification." White Paper, http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.

[59] Jiang, J., Sekar, V., and Zhang, H., "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pp. 97–108, 2012.

[60] Kawaguchi, A., Nishioka, S., and Motoda, H., "A flash-memory based file system," in *USENIX Winter*, pp. 155–164, 1995.

[61] Kgil, T. and Mudge, T., "Flashcache: a nand flash memory file cache for low power web servers," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pp. 103–112, 2006.

[62] Kostic, D., Rodriguez, A., Albrecht, J., and Vahdat, A., "Bullet: High bandwidth data dissemination using an overlay mesh," in *Proceedings of ACM SOSP*, (New York, USA), October 2003.

[63] Kreibich, C., Weaver, N., Nechaev, B., and Paxson, V., "Netalyzr: Illuminating the edge network," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pp. 246–259, 2010.

[64] Kullback, S. and Leibler, R. A., "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, pp. 79–86, March 1951.

[65] Lederer, S., Müller, C., and Timmerer, C., "Dynamic adaptive streaming over http dataset," in *Proceedings of the third annual ACM conference on Multimedia Systems*, MMSys '12, (Chapel Hill, North Carolina, USA), pp. 89–94, February 2012.

[66] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D., "Architecting phase change memory as a scalable DRAM alternative," *SIGARCH Comput. Archit. News*, vol. 37, pp. 2–13, June 2009.

[67] Lee, S.-W., Moon, B., and Park, C., "Advances in flash memory ssd technology for enterprise database applications," in *Proceedings of the ACM SIGMOD*, pp. 863–870, June 2009.

[68] Lee, S.-W., Moon, B., Park, C., Kim, J.-M., and Kim, S.-W., "A case for flash memory ssd in enterprise database applications," in *Proceedings of the ACM SIGMOD*, pp. 1075–1086, June 2008.

[69] Leventhal, A., "Flash storage memory," *Communications of the ACM*, vol. 51, pp. 47–51, July 2008.

[70] Li, B., Qu, Y., Keung, Y., Xie, S., Lin, C., Liu, J., and Zhang, X., "Inside the New Coolstreaming: Principles, measurements and performance implications," in *Proceedings of IEEE INFOCOM*, 2008.

[71] Li, Z., Zhu, X., Gahm, J., Pan, R., Hu, H., Begen, A. C., and Oran, D., "Probe and adapt: Rate adaptation for http video streaming at scale," *arXiv:1305.0510v2*, 2013.

[72] Liao, X., Jin, H., Liu, Y., Ni, L. M., and Deng, D., "Anysee: Scalable live streaming service based on inter-overlay optimization," in *Proceedings of IEEE INFOCOM*, 2006.

[73] Liu, C., Bouazizi, I., and Gabbouj, M., "Rate adaptation for adaptive http streaming," in *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pp. 169–174, 2011.

[74] Liu, X., Dobrian, F., Milner, H., Jiang, J., Sekar, V., Stoica, I., and Zhang, H., "A case for a coordinated internet video control plane," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pp. 359–370, 2012.

[75] Magharei, N. and Rejaie, R., "Prime: Peer-to-peer receiver-driven mesh-based streaming," in *Proceedings of IEEE INFOCOM*, (Anchorage, Alaska, USA), May 2007.

[76] Mok, R. K. P., Luo, X., Chan, E. W. W., and Chang, R. K. C., "Qdash: A qoe-aware dash system," in *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, pp. 11–22, 2012.

[77] Nair, T. R. G. and Jayarekha, P., "A rank based replacement policy for multimedia server cache using zipf-like law," *Journal of Computing*, vol. 2, no. 3, pp. 14–22, 2010.

[78] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A., "Migrating server storage to ssds: Analysis of tradeoffs," in *Proceedings of the ACM EuroSys*, (Nuremberg, Germany), April 2009.

[79] Pai, V., Tamilmani, K., Sambamurthy, V., Kumar, K., and Mohr, A., "Chainsaw: Eliminating trees from overlay multicast," in *Proceedings of International Workshop on Peer-to-Peer Systems*, February 2005.

[80] Park, C., Cheon, W., Kang, J., Roh, K., Cho, W., and Kim, J.-S., "A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications," *Trans. on Embedded Computing Sys.*, vol. 7, no. 4, pp. 1–23, 2008.

[81] Partridge, C., Mendez, T., and Milliken, W., "Host anycasting services." RFC 1546, `http://tools.ietf.org/html/rfc1546`.

[82] Podlipnig, S. and Böszörmenyi, L., "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, pp. 374–398, December 2003.

[83] Qureshi, M. and others, "Scalable high performance main memory system using phase-change memory technology," in *ISCA-36*, 2009.

[84] Rao, A., Legout, A., Lim, Y.-s., Towsley, D., Barakat, C., and Dabbous, W., "Network characteristics of video streaming traffic," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pp. 25:1–25:12, 2011.

[85] Reddy, A. L. N. and Wyllie, J. C., "I/o issues in a multimedia system," *Computer*, vol. 27, pp. 69–74, March 1994.

[86] Saxena, M. and Swift, M. M., "Flashvm: virtual memory management on flash," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, June 2010.

[87] Saxena, M., Swift, M. M., and Zhang, Y., "Flashtier: a lightweight, consistent and durable storage cache," in *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pp. 267–280, April 2012.

[88] Servalli, G., "A 45nm generation phase change memory technology," in *Electron Devices Meeting (IEDM), 2009 IEEE International*, pp. 1 –4, December 2009.

[89] Singleton, L., Nathuji, R., and Schwan, K., "Flash on disk for low-power multimedia computing," in *Proceedings of the ACM Multimedia Computing and Networking Conference*, January 2007.

[90] Stockhammer, T., "Dynamic adaptive streaming over http: standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pp. 133–144, 2011.

[91] Sundaresan, S., de Donato, W., Feamster, N., Teixeira, R., Crawford, S., and Pescapè, A., "Broadband internet performance: A view from the gateway," in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pp. 134–145, 2011.

[92] Tian, G. and Liu, Y., "Towards agile and smooth video adaptation in dynamic http streaming," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pp. 109–120, 2012.

[93] Torres, R., Finamore, A., Kim, J. R., Mellia, M., Munaf, M. M., and Rao, S. G., "Dissecting video server selection strategies in the youtube cdn.," in *ICDCS*, pp. 248–257, IEEE Computer Society, 2011.

[94] Venkataraman, V., Francis, P., and Calandrino, J., "Chunkyspread: Multitree unstructured peer-to-peer multicast," in *Proceedings of International Workshop on Peer-to-Peer Systems*, February 2006.

[95] Wang, F., Xiong, Y., and Liu, J., "mTreebone: A hybrid tree/mesh overlay for application-layer live video multicast," in *Proceedings of IEEE ICDCS*, 2007.

[96] Yu, H., Zheng, D., Zhao, B. Y., and Zheng, W., "Understanding user behavior in large-scale video-on-demand systems," in *Proceedings of the ACM EuroSys*, April 2006.

[97] Zhang, X., Liu, J., Li, B., and Yum, T. S. P., "DONet/Coolstreaming: A data-driven overlay network for live media streaming," in *Proceedings of IEEE INFOCOM*, (Miami, FL, USA), March 2005.

[98] Zhou, C., Zhang, X., Huo, L., and Guo, Z., "A control-theoretic approach to rate adaptation for dynamic http streaming," in *Proceedings of the Visual Communications and Image Processing*, VCIP '12, 2012.